NISTIR 6928

# Concepts for Automating Systems Integration

Edward J. Barkmeyer
Allison Barnard Feeney
Peter Denno
David W. Flater
Donald E. Libes
Michelle Potts Steves
Evan K. Wallace

NIST
National Institute of Standards and Technology
Technology Administration, U.S. Department of Commerce

# NISTIR 6928

## Concepts for Automating Systems Integration

Edward J. Barkmeyer
Allison Barnard Feeney
Peter Denno
David W. Flater
Donald E. Libes
Michelle Potts Steves
Evan K. Wallace
*Manufacturing Systems Integration Division*
*Manufacturing Engineering Laboratory*

February 2003


**U.S. DEPARTMENT OF COMMERCE**
*Donald L. Evans, Secretary*
**TECHNOLOGY ADMINISTRATION**
*Phillip J. Bond, Under Secretary of Commerce for Technology*
**NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY**
*Arden L. Bement, Jr., Director*

# Table of Contents

# 1 Introduction

### *Background*

In the last 10 years, there has been a major change in the way manufacturing entities interact with their customers and each other. The shift is away from centralized engineering, mass production and fixed supply chains, to a new paradigm of distributed product design and engineering, flexible supply chains, and customized solutions based on consumer demand. [58]

The key to achieving flexibility, efficiency, and responsiveness is information – ready access to the right information by the right person at the right time. To be able to capture and deliver the right information, modern manufacturers are consuming information technology products at ever increasing rates [56]. And each of these products improves the capture, flow and delivery of the information that enables some activities in the enterprise to function more effectively. But to achieve agility across the enterprise, information must flow seamlessly from application to application without loss or corruption — from the system that captures the customer's requirements to the system that supports the product designers to the systems that support the parts designers to the systems that support the manufacture and delivery of the parts and the products to the systems that support the maintenance of the products after sale. And every improvement in the capabilities and performance of the enterprise will call for new information flows and the exchange of increasingly complex information.

What we traditionally call "integration" is the engineering process that creates or improves information flows between information systems designed for different purposes. What actually flows between the systems is data, but what is critical to the business process is that all of the right data flows in the right form for the receiving system, and the people who use it, to interpret the data correctly.

Because first-generation manufacturing software was primarily hand-crafted for a particular set of functions in a particular organization, first-generation integration problems involved finding media and data representations for even simple information that could be transferred between these one-of-a-kind systems. During the 1980s and 1990s, massive improvements in computational, communications, and data storage capabilities led to the second generation of solutions to many integration problems. Manufacturers acquired "pre-integrated systems" — a set of function-specific software systems that all worked with a single database based on an integrated business information model for a large segment of the manufacturing enterprise, somewhat tailored to the manufacturer's business processes. These were the Enterprise Resource Planning (ERP) systems, Product Data Management (PDM) systems, Customer Relations Management (CRM) systems, and some others. Across the enterprise, the integration problems were thus reduced to flows between these large vendor-supported pre-integrated systems. And specialized integration firms and technologies arose to solve those problems.

In the last 10 years, however, manufacturing has changed. Many large manufacturing companies have reduced internal costs and increased their market and flexibility by focusing on their strongest product markets and their specialized capabilities, and shifted the "non-core" functions – design and production of many parts and subassemblies, distribution of finished products and spare parts – to supplier organizations. At the same time, *lean manufacturing* placed a premium on time and inventory reduction. Combining these two attributes of the "Quality era" produced a very different business model for many manufacturing organizations – the virtual enterprise. [58]

The new production enterprise is a network of independent companies that shares experience, knowledge and capabilities. Improved communications of many kinds are critical for that sharing to occur effectively, and new software systems have been created to support the inter-enterprise collaborations among the human experts. But converting that network into a business advantage requires the companies' supporting software to exchange information effectively as well. As the design and production of product components becomes increasingly

---

Commercial equipment and materials are identified in order to describe certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose. Unified Modeling Language, UML, Object Management Group, and other marks are trademarks or registered trademarks of Object Management Group, Inc. in the U.S. and other countries.

distributed to suppliers, and the process becomes more responsive to changes in customer demand, the integration problem becomes one of creating information flows among systems across enterprise boundaries. The information that supports the manufacture of an automobile, an aircraft, a home appliance, or an electronics system is now typically distributed over the software systems of dozens, if not hundreds, of separate companies. The shift to a virtual enterprise has turned the second-generation solutions into the components of the virtual information system that supports that enterprise, and that has created a third-generation integration problem.

For companies in the highly diversified U.S. automotive supply chain, for example, lack of information flows, the flow of incomplete, inaccurate, or improperly represented data, and the misinterpretation of received data together impose costs totaling nearly $1 billion each year [57]. That kind of cost is the real "integration problem" that motivates this paper.

### *The NIST study*

Over the last 25 years we have been able to automate the capture and delivery of product engineering decisions to an ever increasing extent, and in the last 15 years, we have begun to be able to automate some of the engineering activities themselves. In a similar way, we have been able to automate the capture and delivery of software engineering decisions for over 20 years, and we have automated some of the software engineering activities themselves.

With this in mind, researchers at the National Institute of Standards and Technology (NIST) have undertaken a project – **Automated Methods for Integrating Systems (AMIS)** – to investigate whether we now have the technology to automate some of the manufacturing software integration processes. The objectives of the project are:

- to identify integration activities that may benefit from automation
- to identify methods and technologies that may be brought to bear on automation of those activities
- to characterize a set of common manufacturing integration problems that may be wholly or partly solved by automated means
- to compare the cost of the automation with the cost of the point solutions, that is, to determine whether engineering for automated integration is easier, better, and/or more feasible than ad hoc engineering of the integrated systems themselves

In characterizing integration problems, the focus on manufacturing software systems is based on the observation that they have characteristics that may render those integration problems different from, and possibly more difficult than, many of the more widely studied "business system" integration problems. In particular:

- Manufacturing, material handling, and inspection systems sense and alter the state of the physical world directly, while most business systems deal with pure information only.
- Engineering data sets are typically large and have complex interrelationships, while most business data sets have relatively simple data structures.

This report documents the results of the first phase of the AMIS project, which addressed the first two objectives: identifying integration activities and identifying methods and technologies that may be used in automating them. Section 2 defines the fundamental concepts and the terms used in this paper. Section 3 identifies the elements of the integration process as performed by experienced software engineers. Section 4 categorizes integration concerns. Section 5 identifies common problems that are encountered in integrating independently-developed software components. Section 6 identifies a number of systems engineering and software engineering technologies that may be brought to bear on various elements of the integration process. And Section 7 presents the conclusions of the first phase and outlines the next steps.

## 2 Fundamental integration concepts

Naively, systems integration is an engineering process for constructing systems. One of the first questions that arises is: What is meant by "automating systems integration"? And in order to answer that question, it is necessary to introduce and define a larger set of terms. This section provides definitions of the fundamental concepts for systems integration and the terms used to describe them. Figure 1 is a UML [80] "static structure diagram" showing the relationships among the principal concepts defined in this section.

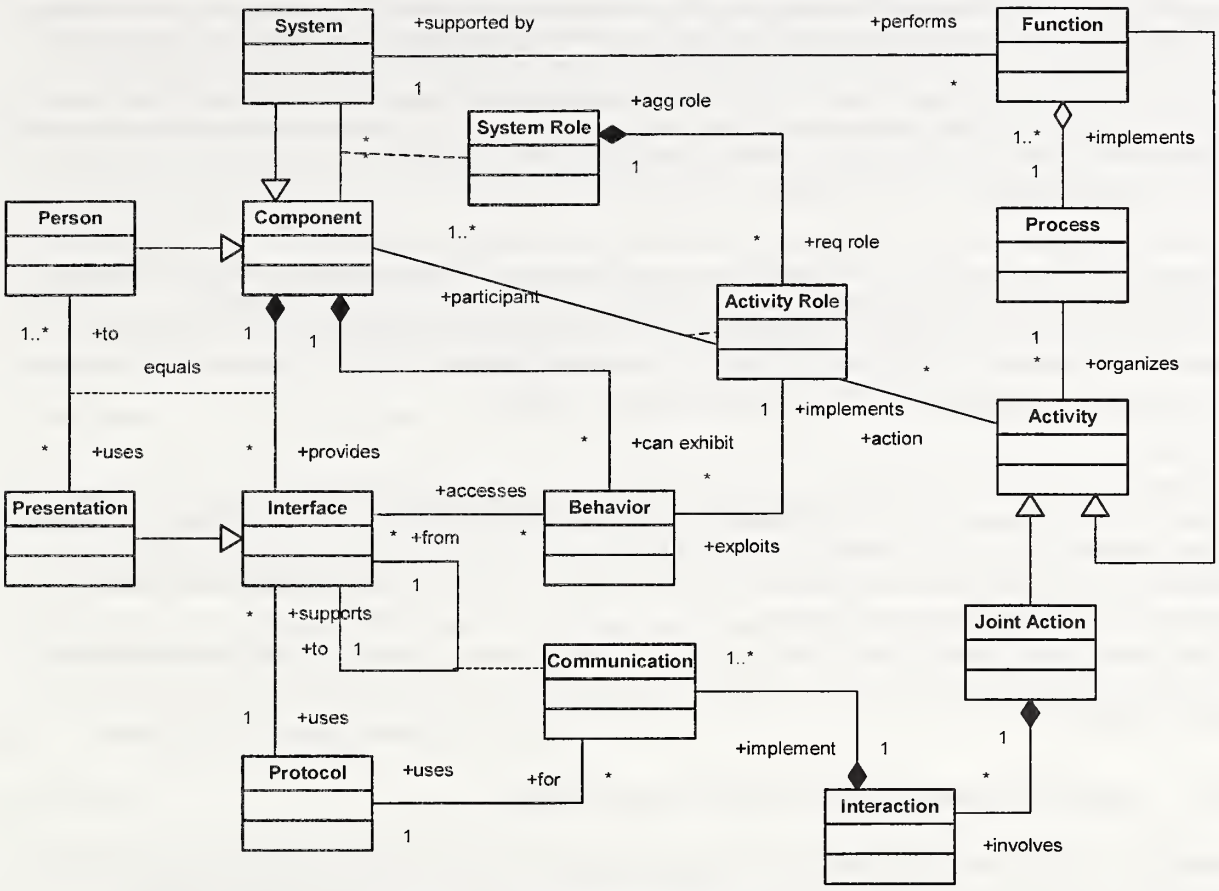


Figure 1 Fundamental integration concepts

## 2.1 Systems, agents and components

***System*** = a complex of software, hardware, and human resources that jointly accomplishes one or more business functions. A system may be pre-designed or arise ad hoc by action of one or more of the participating human or software resources.

***System design*** =

(1) a specification of the structure of the system, including the component view, the information view, and the computational process view.

(2) a breakdown of system functions into sequences (graphs) of subfunctions assigned to nominal component subsystems, coupled with a specification for the information and materials that must be available at the component interfaces in order for the subfunctions to be accomplished.

***Component*** (or ***agent***) = a system that plays a particular role, by providing some or all of its functions as subfunctions, in the functions of a larger system, either pre-designed or ad hoc.

***System-specific component*** = a system designed to be fitted into a specific set of named roles in some larger system for which the design, and therefore the interface specifications for the component, are known a priori.

***System-specific component design*** = specification of a subsystem to play the role of a particular component in a specific system.

***Reusable component*** (or ***strange agent***) = a system designed to be fitted into a specific set of named roles in some larger systems to-be-designed (or to-arise), where the larger system designs are not known a priori and the interface specifications for the component are defined based on presumptions of the interface requirements for the target roles. The larger system design is expected to incorporate this component in one or more of the anticipated roles and to specify the other components providing the roles this component is designed to interact with. The further expectation is that these other components will have been designed to use the published interfaces of the reusable component in its intended roles. In reality, however, this design may happen after the fact and the component may be used in roles somewhat different from those anticipated.

***Reusable component design*** = specification of a subsystem that exhibits a set of behaviors (can perform a set of sub-functions) that are useful to an undefined system with some nominal business functions. This includes the specification of the expected information flows and the explicit interfaces used to activate the behaviors and enable the information flows. The design of a reusable component often explicitly includes capabilities by which the reusable component can be "configured" to exhibit any of several variations in its principal behaviors and interfaces, by setting certain parameters, some at build time and some at runtime.

Examples:

(1) an ERP software package installed on a server computer system and configured for use in a particular enterprise, together with its terminals and the trained human operators, is a pre-designed *system*. The ERP software package alone is a *reusable component*. The terminals and trained operators are *system-specific components*. An add-on third-party software package that automates an additional business function is a *reusable component* that can be added to this system to make a larger system.

(2) the interaction of a buying agent for one company and a selling agent for another company in negotiating the terms and conditions of a sale is an ad hoc *system*. The buying agent and the selling agent are *strange agents* (or *reusable components*).

## 2.2 Functions, processes, resources and roles

***Behavior*** = what the system/component does — the activities of a system component as observed by itself, by other components of the system and/or by the system engineer, via its effect on its environment and/or through measurable attributes of the component itself. Behavior of an information system element can be influenced by environmental conditions, past actions, or configuration.

***Function*** =

(1) the part of a component's behavior that satisfies some objective or purpose. If one models behavior as a set of transitions of state in the observed environment, then function can be modeled as the subset of those transitions that serves the purpose of the system. That set of effects can be seen as implementing more than one function, if different subsets of those effects are considered useful with respect to some target business process.

(2) the result that behavior achieves, particularly when the actual behaviors are not important.

***Process*** = the "decomposition" of a function — the temporal or logical ordering (a graph) of events/activities that accomplish a function or functions.

A process is said to be ***automatic*** if there is no human involvement in its initiation or execution.

A process is said to be ***automated*** if it occurs with no human intervention after initiation, except in extraordinary circumstances. (That is, the automaton may permit human intervention to override its automatic decisions, and it may request human intervention if it detects a situation for which it has no decision-making algorithm.)

A process is said to be ***semi-automated*** (or ***computer-assisted***) if part of the process is automated, but human intelligence is employed during the process in making the important decisions and accomplishing the goal.

A process is said to be ***manual*** if there is no significant computer assistance in its execution.

Behavior, function, and process are all views of the operation of any machine:

- Behavior is the "action"/"execution" view of operating the machine — what the resource is doing.

- Function is the "state" view of operating the machine — the effect on the environment.
- Process is the "thread" view of operating the machine — how the function is being accomplished as a partially ordered set of actions by different resources.

***Resource*** = any person, device, software system, material, or information set that performs, or is used in the performance of, one or more functions.

***Active resource*** or ***Actor*** = a resource that directly performs one or more functions, and (typically) makes decisions in the performance of the function.

***Agent*** = an active resource that is primarily a software system.

Note – In the literature, *agent* is sometimes used as a synonym for actor, and it is often used with much narrower interpretations with respect to autonomy or technologies used, but there is no accepted definition.

***Passive resource*** = material, equipment, or information used by active resources in the performance of a function.

***Role*** =

(1) in an activity (*activity role*): the participation of a nominal resource in the activity.

(2) in a system (*system role*): the set of all activity roles played by a given resource in the processes corresponding to the set of functions supported by the system.

In either case, a role is characterized by required behaviors, and by characteristics that imply the ability of a resource to exhibit those behaviors.

## 2.3 Integration and communication

***Integration*** =

(1) a *state* of a set of agents that enables them to *act jointly* in one or more sub-functions of a function of a larger system. The system may be pre-designed or ad hoc, and the agents may be system-specific components or reusable components. The agents must exhibit behaviors consistent with the accomplishment of the system function and *communicate* to the extent necessary to accomplish the *joint action*.

(2) a systems engineering *process* – a design/development activity that

- identifies how *joint action* of two or more independent agents enabled by particular *communications* can improve a business process,
- results in a larger *system* that can accomplish the end goal of improving the business process, and
- includes assignment of *roles* in functions of the larger system to nominal *resources*, determination of the suitability of specific resources to fulfill those roles, the configuration or modification of the resources to enable them to perform those roles, and the creation or acquisition of additional resources where needed.

***Joint action*** = any productive way in which multiple active resources participate in accomplishing a single function, including coordination, collaboration, cooperation, unwitting assistance, witting non-interference, and even competition. The critical factor is that the interaction among the resources acting jointly involves some *communication* on the part of each participating actor, and the joint action accomplishes some part of a function of the larger *system*.

***Information*** = data with sufficient semantics of the producer preserved in interpretation by the consumer. Some bodies of information are *communications* that enable integration and some bodies of information are *passive resources* that are used by one or more active resources in the performance of certain functions. And some communications may be archived, i.e., converted to passive resources for the future.

***Communication*** =

(1) any operations that result in a flow of information from one actor to another, including rendering of internal information into data for transmission, flow of data by any of several *mechanisms*, and interpretation of received

data into internal information. The flow occurs by specific actions on the part of two (or more) actors who may or may not be directly aware of each other.

(2) the exchange of data between actors in the performance of a joint action, with sufficient semantic content preserved (through the data flow and data interpretation operations) that the intended joint action can occur and achieve the intended effect.

Note — Communication (1) does not require "language." It is any mechanism that causes information to flow, including physical signals and physical effects that have no "data representation."

Note — One actor may communicate with another for the purpose of obtaining specific behaviors of the other party that are relevant to a function the actor is performing. Alternatively, the systems engineer may cause communication between two actors for the purpose of obtaining specific behaviors from either or both, but either or both of the actors may be unaware of that purpose. That is, the purpose of communication between software components is to influence behavior, witting or unwitting. There is no other purpose to communication.

***Communications mechanism*** = the way in which information flows between resources, as characterized by the fundamental behaviors involved in the communication operations. The mechanism is a higher-level abstraction than the detailed rules for the information flow, which are called *protocols*. All of the following mechanisms are distinguished, in that they involve distinctly different behaviors of the communicating resources:

- file or document
- shared database
- procedure call or operation invocation
- messaging or streams (asynchronous one-way transmissions, often supported by queuing and retrieval)
- blackboard (publish and subscribe, shared memory, etc.)
- signals (physical signals and software events)
- human interface mechanisms (such as display, keyboard, mouse, all of which are distinct mechanisms)

Note — Many of these techniques are distinguished in the "application layer" of the Open Systems Interconnection (OSI) Reference Model [116].

## 2.4 Automated integration

***Integration automation*** = Automation of the *integration process* or some aspects of it, i.e., full automation of some processes in the design and development of a larger system. In the narrowest case, human engineers identify the need, specify the goal, see the joint action required, and assign the roles to the resources. Software automates the communications that enable the joint actions.

***Self-integration*** =

(1) Automation of some aspects of the design and development of a larger system by action of one or more of the reusable components themselves. Self-integration is an integration process in which the software agents adapt themselves to assist in enabling the joint action and/or the communication.

(2) More narrowly, integration in which the *technical* requirements for the communication are automatically resolved by the communicating agents, as distinct from being pre-resolved in the design of the system or agent and pre-programmed into the components. The technical requirements are:

- common technology for the transmission and reception of the data
- common protocols for the exchange, i.e., agreed-upon rules for the elementary interactions
- common data structure and representation

Note – Self-integration is a subtype of integration automation that requires automation of the enabling communications by action of the agents themselves, as distinct from automation by action of a third-party tool that builds a bridge or set of wrappers and runs to completion before the agent interactions occur.

***Self-integrating system*** = a system that is able to reconfigure itself and/or its communications to cooperate with other systems to meet new requirements (of larger systems). In general, the ability of a system to meet new

requirements depends on those requirements being met by designed-in behaviors of the system (which may themselves be configurable) or by modified behaviors developed by designed-in "learning mechanisms".

***Self-describing system*** = a self-integrating system that is connected to other systems in a network environment, advertises its capabilities (possible roles), and automatically determines the capabilities (roles) of the other systems from their advertisements. Capabilities can be stated generally or with respect to some built-in functions or goals of the self-describing system or of the putative larger system that incorporates all the self-describing systems on the network. A self-describing system formally communicates (in some way) the set of behaviors it can/will exhibit, and exhibits them as required by actions of other actors and its own comprehension of its role in the larger system. The purpose of self-description is to facilitate reuse of components.[1]

# 3 Integration process elements

The object of the integration process is to get separately designed resources to work together to accomplish some end goal. This section identifies the principal activities that can take place in an integration process, and the engineering tasks (process elements) that may be included in those activities. An overview of the entire integration process (which is really the systems design process) is shown in Figure 2.

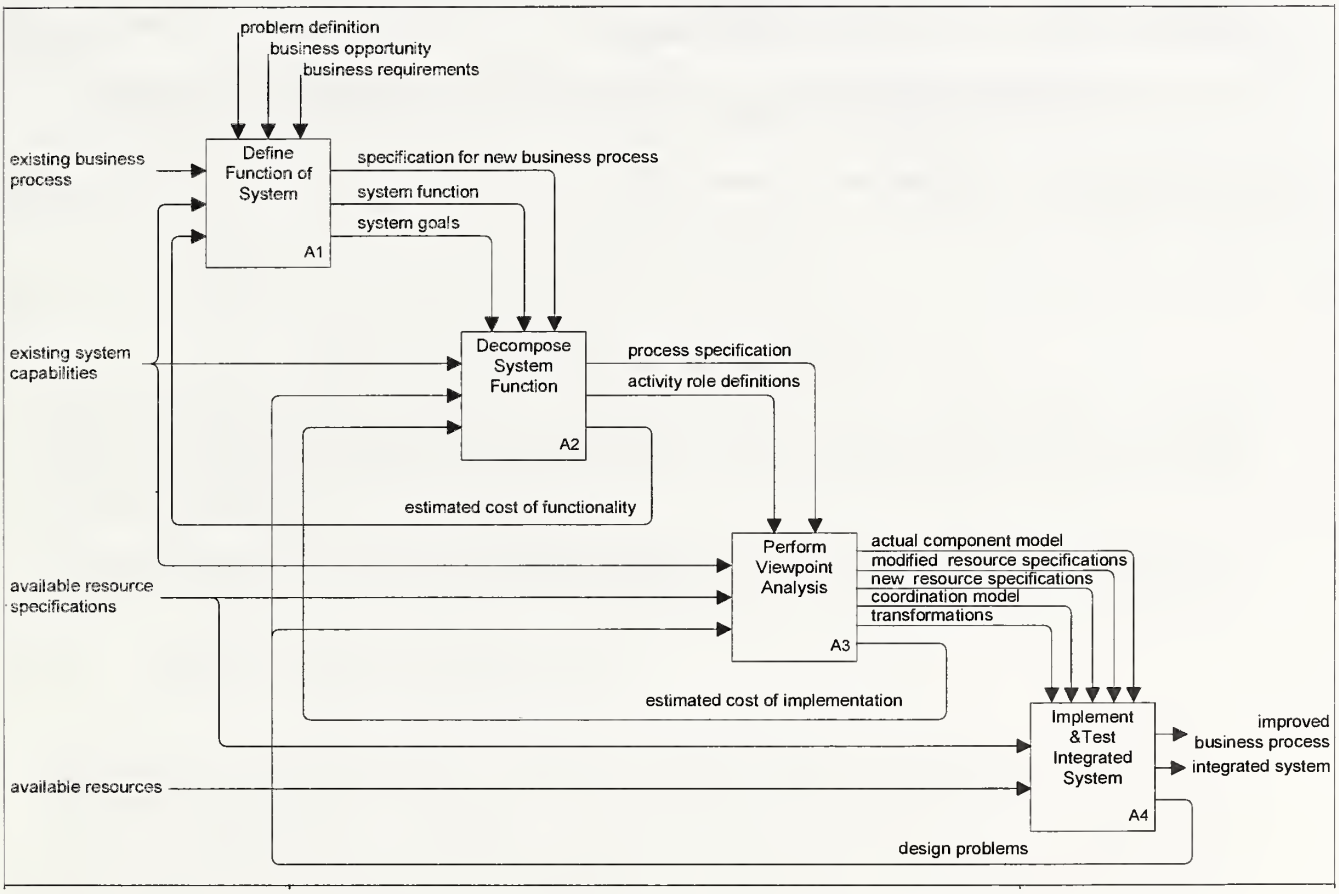


Figure 2 System Integration Process Overview

[1] One should not confuse self-describing systems with *autonomous* systems, which are often self-describing. An autonomous system has its own built-in goals, while a reusable component only has behaviors presumed to be useful to the goals of a larger system. An autonomous system self-describes so as to cooperate effectively with other autonomous systems (that also self-describe) in interactions that help it achieve its goals.

"Engineering is a combination of top-down and bottom-up reasoning." [87] And there are two corresponding approaches to formulating an integration problem.

The top-down (or "function-driven") approach is:

- First: What is the business process I have, or would like to have?
- Then: What resources do I have available, and what do they do that is useful in supporting that process? What can I use directly? What can I change? Where are the gaps?

The integration problem is to define the changes and fill in the gaps.

It is important to distinguish function – what the system is trying to accomplish – from behavior – what the component actually does. It is the behavior of the component that is most important in integrating it into a larger system; what function it was designed to serve may influence the degree to which that behavior can be usefully exploited.

The bottom-up (or "data-driven") approach is:

- I will continue to use systems X, Y and Z to support certain elements of my business process.
- If I could get certain information into system X, I could improve my business process in a number of ways, by making better information available to several decision makers, both human and automated.
- System Y has part of this information, and system Z has the rest. How can I get the data out of systems Y and Z and into system X?

The integration problem is to make that information flow happen.

How we state needs is often related to the machines we are using or expect to use. Sometimes the need originates from using those machines in our process and sometimes it is easier to state the problem in terms of those machines.

These different approaches to integration are reflected in Figure 3.

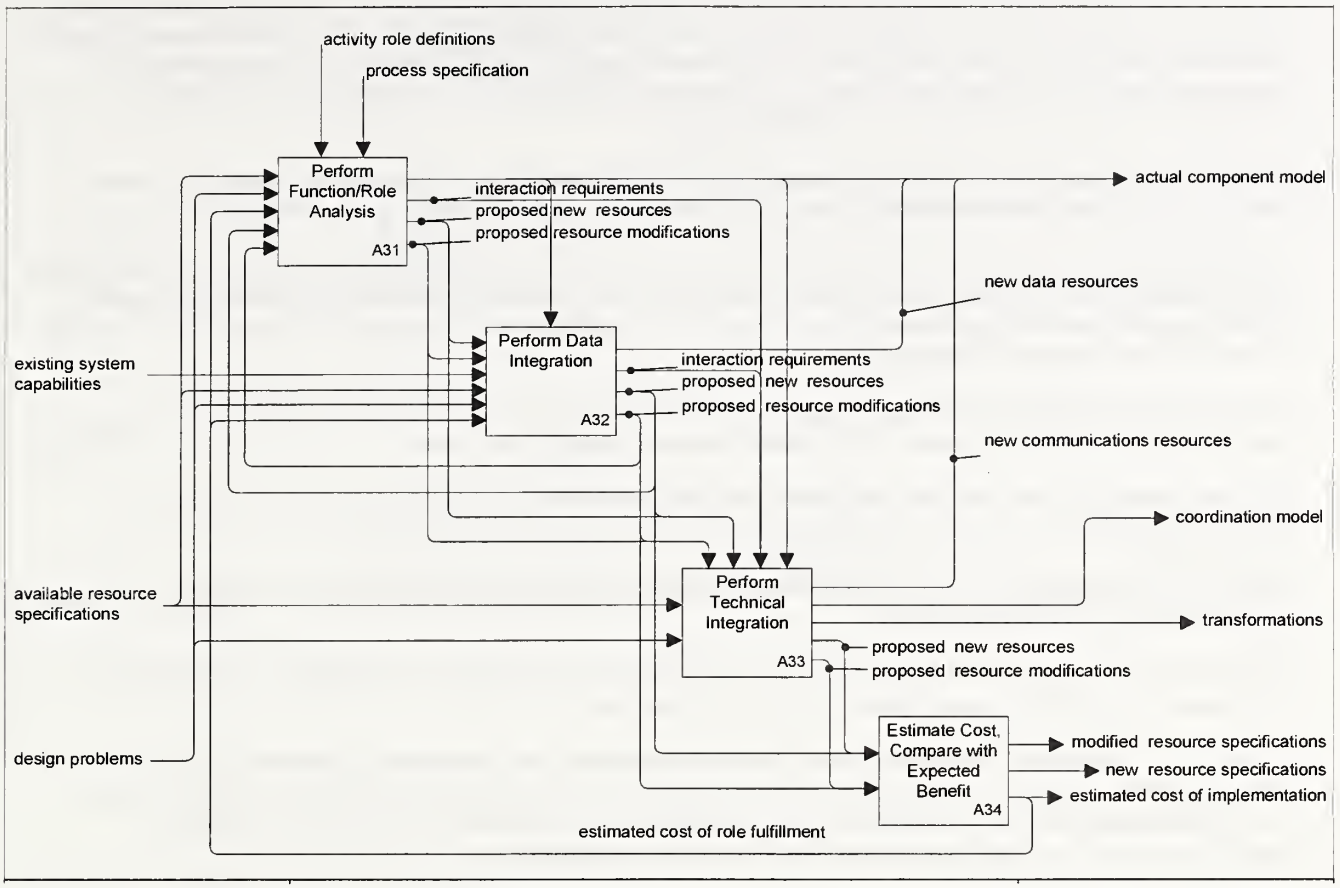


Figure 3 Integration Process: Viewpoint Analysis and Design

But these approaches tend to be flexible with respect to both requirements (top-down) and specifications (bottom-up), and the actual design process is often cyclic refinement. There is a continuing tradeoff between what is ideal and what is feasible and cost-effective at several levels of design. Ultimately "what I need" becomes "what would be useful that can be built."

As a consequence of these variations in the engineering approach, any given integration process may involve only some of the activities discussed below and will organize the process elements as needed for the problem at hand.

## 3.1 Define the function of the system

The first step of any engineering process is to define the problem to be solved and to make the decision to invest some resources in designing a solution. This step in some sense precedes the engineering process itself. In the case of systems integration projects, this first step is to identify an opportunity for improving the performance or extending the capabilities of the organization in some business activity by improving the automation of the business processes. This may mean the increased automation of an existing manual or semi-automated process, the addition of new capabilities to the business systems, the replacement of business systems, and/or the replacement of a business process with a new business process that takes advantage of additional capabilities. In some cases, an entire business process can be automated; in others, the process will be semi-automated. The new system may reduce the amount of human involvement required, improve the information available in making business decisions, increase the market or volume that can be accommodated, reduce organizational response time, improve the quality of results, and/or reduce the errors, losses, and costs in the performance of the process.

In some cases, the engineering action is initially a study of whether a given business process is practicable or cost-effective; in others, it is based on a purely business decision that requires the systems of two organizations or trading partners to work together; and in still others it is the pursuit of an idea for taking advantage of an opportunity to improve capabilities or reduce cost. That is, the rationale for the integration activity is always a management decision, which may or may not have a technical foundation.

The first engineering task is to define the business process to be supported by the system, and define the goal and functions of the system to support that process. The basic elements of this task are shown in Figure 4.

In general, the *goal* of a system is the set of functions it must be able to perform in support of the business process. But for some systems, the goal is to minimize the value of a "cost function" whose inputs include measurements of certain states of the world as perceived by the system, or even more narrowly, to reduce the friction between existing systems in supporting the intended business processes.

The integration problem may be stated in terms of the business process to be supported, or in terms of the joint actions of specific components that will be required in order to improve the process.

The new business process, and the corresponding requirements for the target system, may have to be revised if subsequent design efforts reveal that the cost of the original target will be unacceptable.

Inputs: business opportunity, business requirements, existing business process, analysis of capabilities of existing system(s), budget, and estimated costs

Outputs: a specification of the business processes and/or the corresponding goals and functions to be supported by the system, i.e., a high-level specification of what the system does.

Automation: This process is and will remain a semi-automated process for the foreseeable future, in that the decision-making requires human system engineers, but many of the inputs and outputs might be formally captured in machine-interpretable form. Specifically, what is required for the automation of later activities is that the goals/functions of the system must be specified in machine-interpretable form.

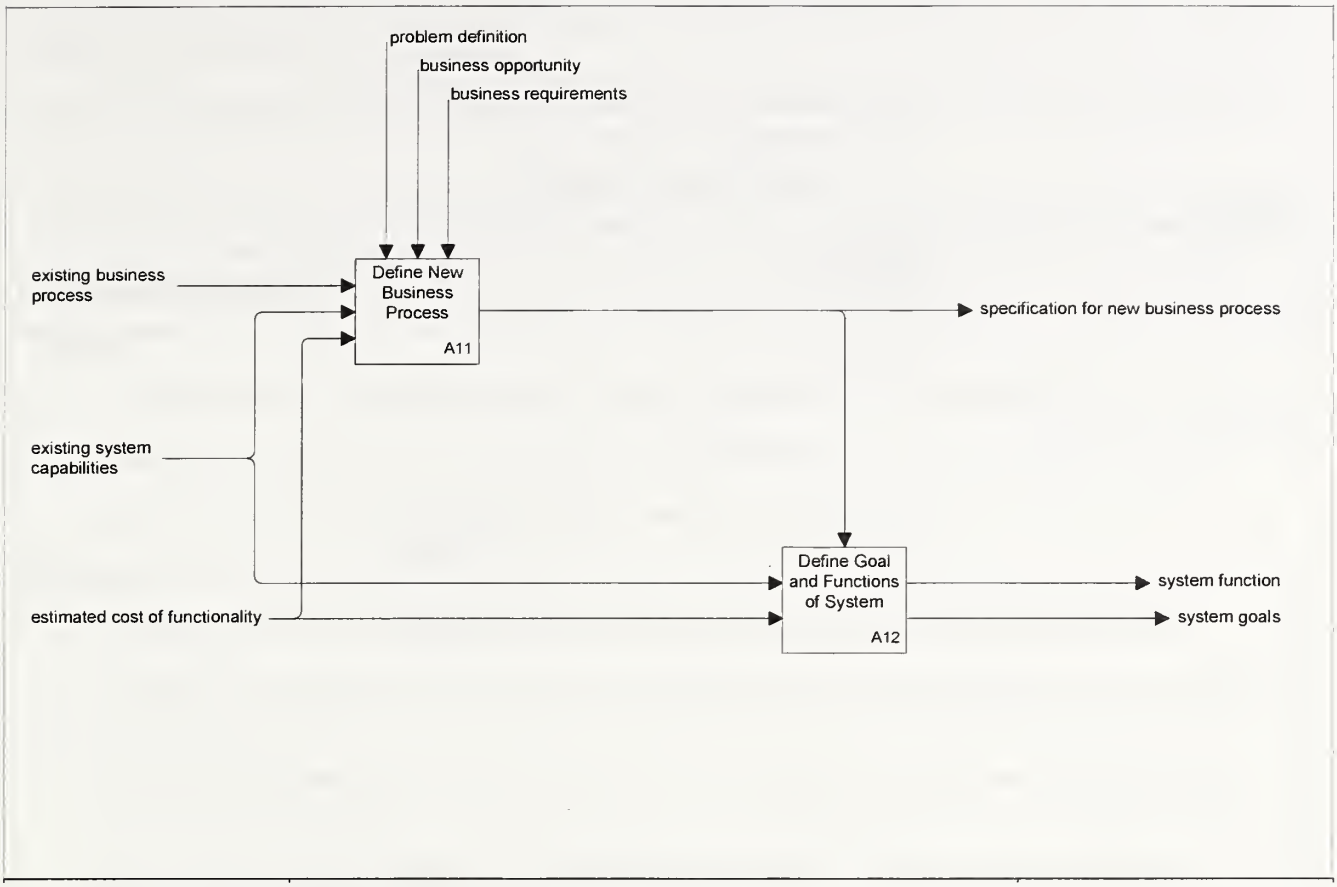


Figure 4 Define the Function of the System

## 3.2 Identify the joint actions, resources and capabilities required

For each function (or goal) of the system, define the process of accomplishing that function as a graph of tasks (operations, sub-functions) to be performed by nominal or specific actors, some jointly, some independently. All of the steps in the process must be traceable to the required end results, although some steps may be enablers for others. The basic elements of this task are depicted in Figure 5.

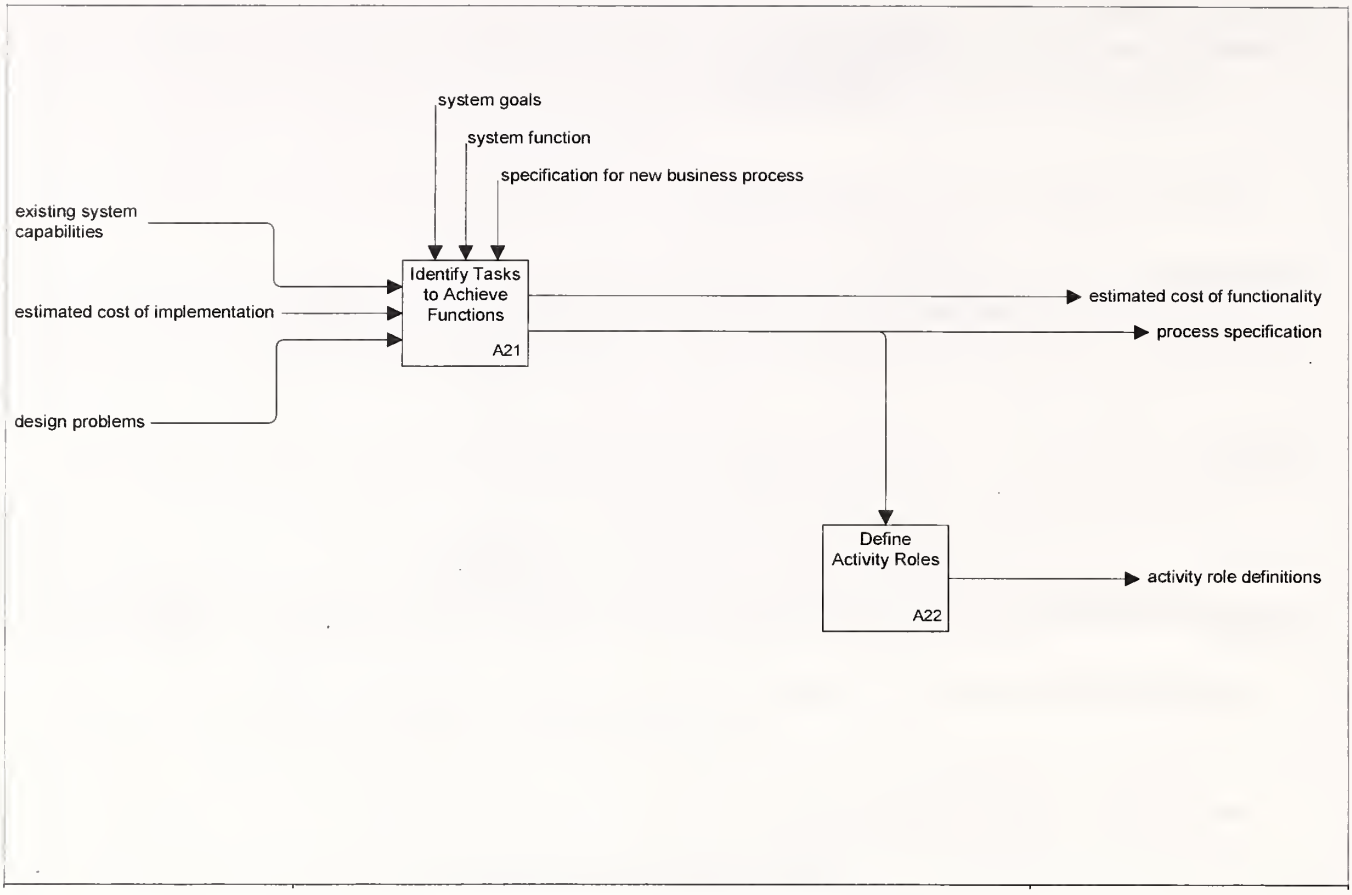


Figure 5 Identify Joint Actions and Define Roles

For each task, define the use of resources in specific *roles*. A role is characterized by the required behaviors of the resource and by characteristics that imply the ability of the resource to exhibit those behaviors. A role may be specified by identifying a specific resource that is available, or a class of resources from which one or more resource instances will be chosen, and the set of capabilities/behaviors of the resource that are required for the task. Alternatively, a role might be characterized entirely by the capabilities/behaviors required for the task, without actually specifying the resource or type of resource that is wanted.

While integration tends to concentrate on the roles of active resources in the system tasks, the roles of passive resources – materials, tooling and adapters, communications appliances, reference data sets and databases, execution scripts, etc. – may also be critical to the success of the resulting system. Integrating the passive resources into the system may also present technical problems. So it is important not to overlook the characterization of the passive resource roles required for the tasks.

Among the activities required for active roles will be the joint actions in which the resources performing two or more distinct roles will interact. Like other actions in the process, it is the target function that motivates the joint actions of components and therefore the interactions. Whether the actors themselves must be cognizant of the goal for their interactions depends on whether they must know the goal in order to cooperate in achieving it.

In a joint action, requirements derived from the function that the joint activity is intended to serve must be allocated to each such interaction. At the same time, additional requirements derived from the need for the two actors to interact become part of the requirements for the joint activity. These additional requirements are called *co-ordination requirements*. They may be imposed separately on the participating actors, according to their roles. In particular, co-ordination requirements include the requirements for communication between the participating actors. And the required resource capabilities associated with roles in joint actions will include the ability to perform the

communications necessary to enable the joint actions. These communications are usually the primary focus of the detailed technical aspects of the integration.

In general, this problem has multiple solutions. In most cases, the solution-of-choice will be strongly influenced by the resources available. So this activity may be performed jointly with 3.3.

This process may also be re-iterated as the further design of the integrated system reveals that the chosen breakdown of system tasks is expensive to implement or inadequate to support the target business process. This process may also reveal that some desired functionality will be very difficult or expensive to achieve, and therefore necessitate rethinking the system function requirements.

Inputs:

- A specification for the revised business process, and the corresponding system functions and goals
- A description of existing system capabilities and the capabilities of other resources deemed to be available
- Estimates of cost and adequacy of previous process designs and role assignments

Outputs:

- A definition of each system function as a process specification – a graph of tasks with associated roles
- A definition of each role, either as a behavior to be exhibited or as a set of characteristics that imply the ability to exhibit that behavior
- A definition of the interactions required among the roles to accomplish the function, and the corresponding coordination requirements
- Estimates of the cost of achieving the desired functionalities, based on examining the costs of alternative process specifications and role assignments

Automation: If the goal and the system were specified with enough rigor in a fully-defined, unambiguous context, it might be possible to derive solutions in a procedure similar to theorem-proving. But in most cases, the important decisions in this area will require the expertise of human systems engineers, supported by tools that capture the design of the processes and the specifications for the roles, and tools that support costing and evaluation of alternatives.

## 3.3 Assign system roles to resources

Identify the behaviors, or the sets of required characteristics, over all activity roles of a nominal resource. That set defines an overall role for the resource in the overall functionality of the system – a *system role*.

Examine the behaviors and characteristics of the available resources, determine which resources are suitable for which system roles (see 3.3.1), and assign each system role to a specific resource. Some resources may fulfill more than one identified role. Unfortunately, this task may have to be repeated if it is subsequently discovered that certain resources do not actually have the full capabilities ascribed to them. In particular, there may be certain impediments to their successful interaction with other system components (see Section 5) that make them unsuitable for a role to which they are assigned.

Identify the roles for which there are no suitable resources available. Where appropriate, create the specifications for configuring or modifying existing resources, or building or acquiring new resources, to fulfill those roles.

Compare the estimated development cost with the expected benefit from the system, and determine whether to proceed with the design for the integrated system or to reconsider the system functionalities and system process definitions. This task may be repeated when a more detailed design results in more accurate estimates of the development cost.

An overview of the basic elements of this task is presented in Figure 6.

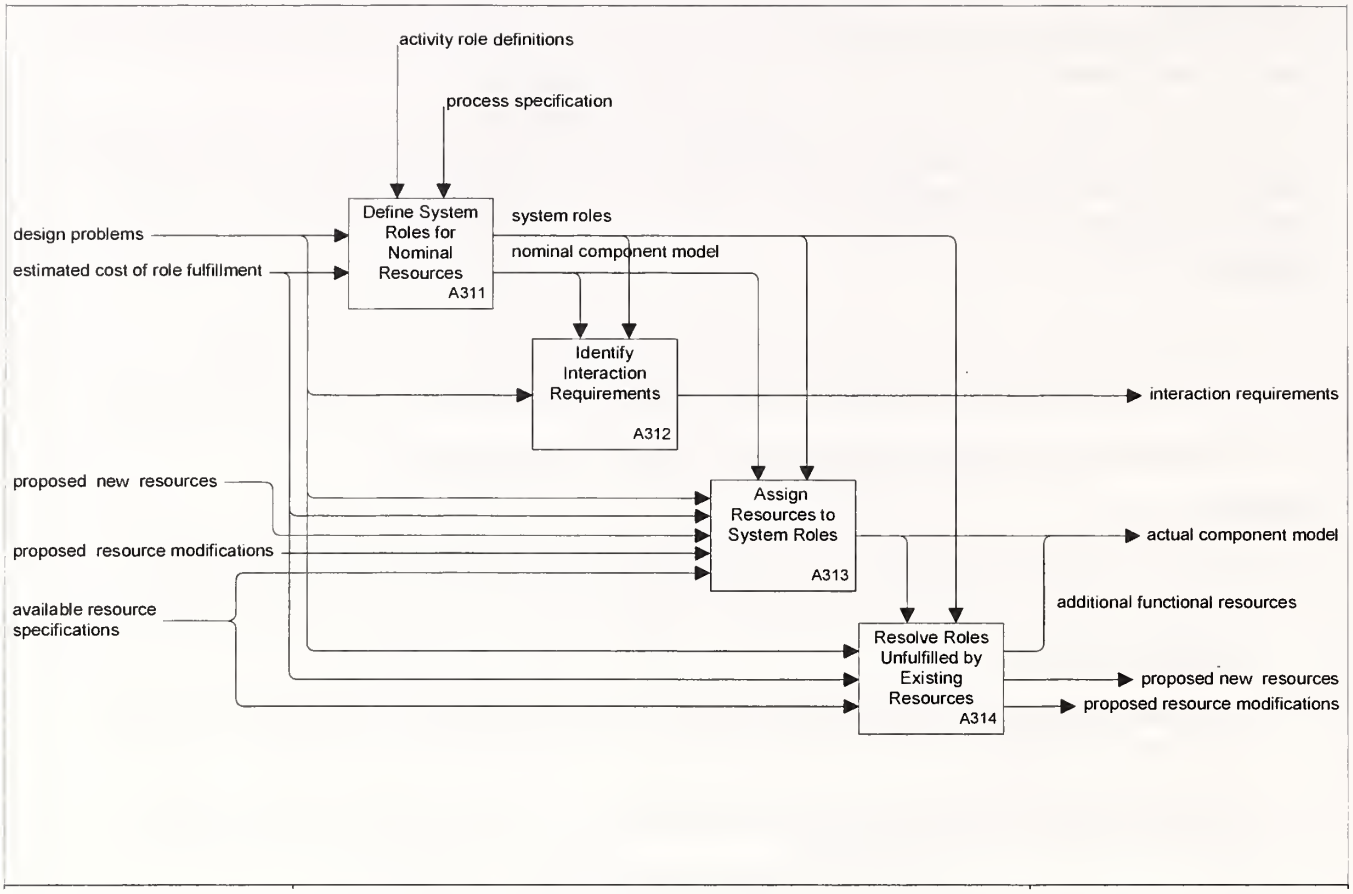


Figure 6 Assign Roles to Resources

In general, this problem has zero or more solutions. In many cases it will have no true solution, but several adequate approximate solutions. Some problems simply do not admit of automated integration, and the most common integration implement is a human-designed adapter that preserves as much semantics as the human designer understands to be appropriate (and possible).

Inputs:

- Process specifications and activity role definitions
- Specifications for available resources, and corrections to their behavioral descriptions based on further analysis and experience
- Estimated costs of role fulfillment from detailed design and implementation activities

Outputs:

- A nominal component model of the system, representing the system as a set of nominal active and passive resources, characterized by distinct system roles, that together support all functions/goals of the system
- For each such nominal resource, a definition of its system role: a set of specific behavior requirements and communications requirements that supports its assigned sub-functions

- For each such system role, the identification of the actual resource that will fulfill that role, possibly including a set of specifications for its configuration or modification, or the specifications for each additional resource or capability that must be acquired or developed, in order to fulfill the role

Note: this will be a reasoned choice among several possible solutions, or in many cases "near solutions."

Automation: This requires formal characterization of behaviors of active resources, both expected and actual. It also requires formal characterization of the communications among them at some semantic level. It may or may not include formal characterizations of the communications at a technical level. (These levels are further discussed in section 4.) Given formal characterizations of the resources, and formal characterizations of the required capabilities, the rest of the process is a "semantic matching" problem. It may be possible to automate the solution to such a problem by a combination of deductive methods and optimization methods. It is more likely that such mechanisms can be employed to solve parts of the problem with considerable interaction with human engineers. This is an area for further research.

### 3.3.1 Suitability of a resource for a role

"Integration of a resource" means matching a resource to a role in the target system and, if necessary, modifying the resource or providing supporting resources to allow it to play that role. Any other compatibilities among resources in the system are irrelevant to the given integration problem, although they might be important to some future intents for which the requirements are not yet well-defined.

Suitability of a resource for a role is based on the *behaviors* the resource exhibits or is capable of exhibiting as compared with the required behaviors of the role in the system. With respect to the suitability of a resource for a role, we can distinguish:

- *incompatible* resources, which have behaviors that conflict with the requirements for the intended role or other operations of the system and cannot readily be modified to avoid the conflicts

- *integratable* resources, which have capabilities and behaviors that support the requirements of the intended role, and have no behaviors that conflict with the requirements of the role or with other operations of the system

- *partly integratable* resources, which have capabilities and behaviors that support the requirements of the intended role, but are also intended to fulfill or use other roles that are in conflict with the requirements for the system. That is, partly integratable resources are compatible with the role, but they are compatible with the system *only* in a subset of the concepts that the resource and the system both support, so that some of the behaviors/capabilities of the resource must be avoided or suppressed

The suitability of a resource for a role may also be influenced by interoperability, although that is less often an option in practice. But in defining the system:

- An *integrated* resource must be both integratable into its role and interoperable with each of the resources with which that role requires it to communicate. Such a resource can become a part of the system without modification.

- A *partly integrated* resource must be partly integratable into its role, interoperable with each of the resources with which that role requires it to communicate, and prevented from exhibiting its counterproductive behaviors. Such a resource can become a part of the system when the suppression of, or a work-around for, the undesirable behaviors is in place.

### 3.3.2 Interoperability of resources

Required behaviors of a resource in a role typically include requirements for communicating with resources playing other roles in accomplishing certain activities. Integratable and partly integratable resources must be able to communicate in a way that is consistent with their system roles, but they may or may not be able to communicate with other resources using a given technology, protocol, or data representation. That is, the *functional* requirements

for these communications are characteristic of the role, but the *technical* requirements for these communications — technology, protocol, data representation — depend on the *specific resources* involved in a given communication, and are not usually intrinsic to the role. For example, a subsystem could perform the same function in different system activities and use different data representations to communicate similar information to different partners in those activities.

For each distinct *pair* of roles that must communicate, the corresponding resources can be further classified with respect to the technical requirements as:

- *interoperable* – able to use the same technologies, protocols, and data representations; or
- *non-interoperable* – able to communicate the common intent, but only with some difference in the technologies, protocols, or data representation conventions.

Note – In many integration scenarios, several of the major resources involved are fixed a priori, and there is a need for specific communications between the roles these major resources play. Most of the integration problem in those cases is overcoming the lack of technical interoperability.

## 3.4 Data integration

The typical industrial system integration problem is mostly bottom-up. That is, most of the resources involved in the new system have long been chosen and installed and are known to be integratable into the roles chosen for them, because those roles were chosen with the capabilities of those resources in mind. The resulting integration problem is then primarily a matter of developing the interfaces and intermediaries necessary to provide the necessary data and cause the specific resource to execute the required functions. But this same situation also arises at the end of a top-down process when reusable components are involved.

The data integration process begins with a role definition and resource assignment activity at a "lower level of abstraction" than the business process. It is specific to enabling information flows for primary functional resources that have already been chosen. In general, this part of the integration process then takes the following steps, which are depicted in Figure 7:

- For each task Y that is required of sub-system X, identify the information requirements for subsystem X to perform function Y.
- Find the data that satisfies those information requirements among the resources (humans, subsystems, and repositories) potentially available to the integrated system.
- Identify gaps between the available data and the required information, including both missing information and information that requires semantic derivation from available data.
- Where necessary, specify the requirements (roles) for new resources that will provide the missing information, by possessing it, by converting manual repositories to machine-readable form, or by capturing it from human agents or sensory devices.

Finding the required information properly includes:

- reasoning backward from the semantic model of task Y in system X to the required (quality of) system X inputs to achieve the desired level of quality in the system X performance of task Y, (that is, one needs to know the impact of the availability and accuracy of certain inputs on the behavior of the system in performing the task)
- then reasoning backward from the set of required inputs and forward from the sets of data available from other component resources to an algorithm for deriving the needed inputs from the available data (a set of functions, some of which are the identity mapping), and the corresponding information flows required

Note – An important characteristic of data sets, which has both semantic and technical elements, is the notion of "data instance" or "object identity." That is, in finding the data needed to perform a function for a given set of business objects, it is important to share the *identification* of those objects and the data sets that describe them. In addition to agreeing on the type of the object and the meaning of the properties exchanged, interacting resources must agree on identification conventions to determine which specific object is being discussed.

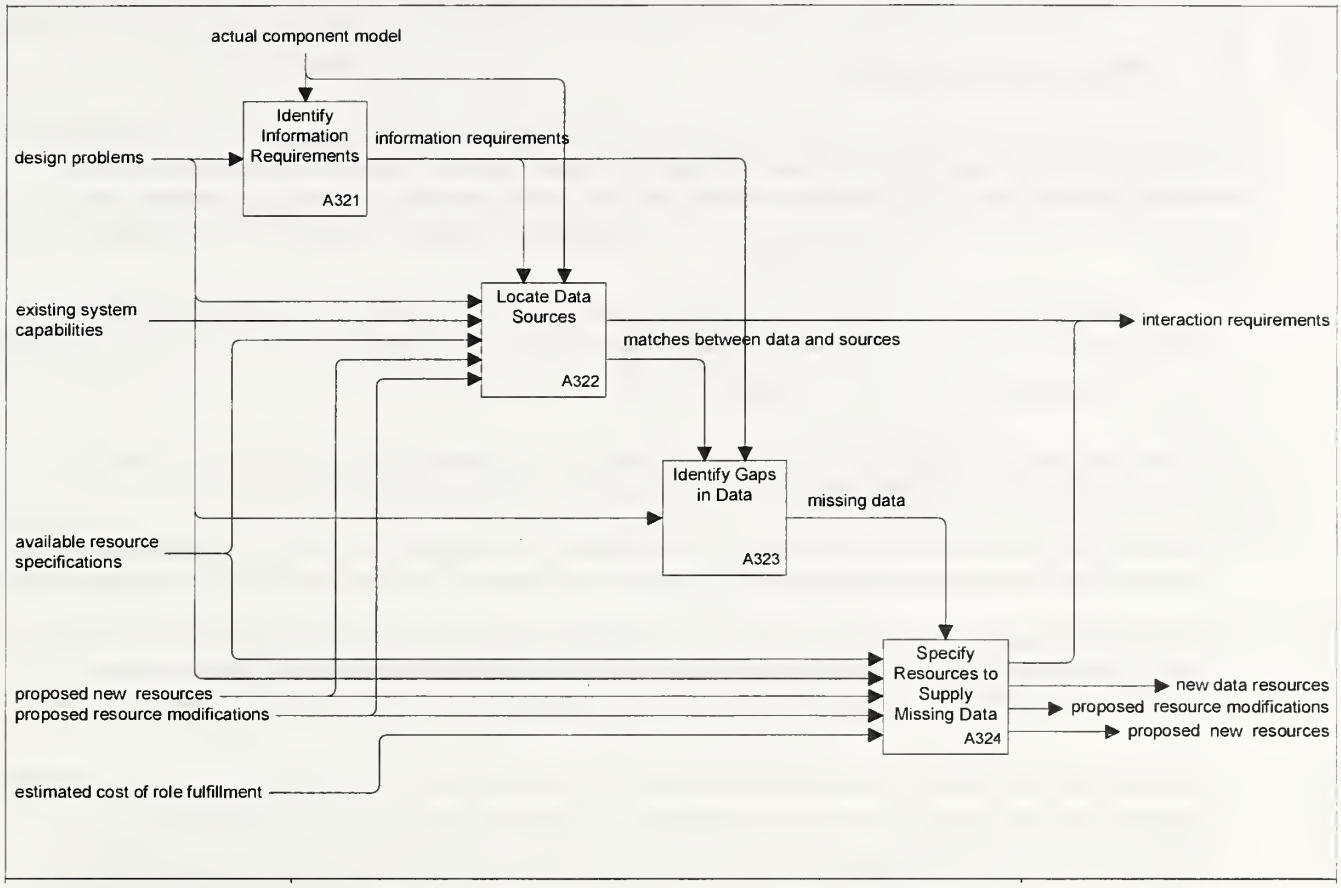


Figure 7 The Data Integration Process

Inputs:

- the actual component model for the system – the functional resources and their roles
- specifications for existing and available resources and new or modified resources proposed to meet the functional requirements, and corrections to those specifications with regard to data availability, completeness and accuracy, based on further analysis and experience
- estimates of the cost of supplying the additional or modified resources proposed to meet data requirements

Outputs:

- specifications for component interactions required to enable data flows
- proposed new resources and resource modifications needed to meet data requirements

Automation: Automating this task requires a formal characterization of the information required for each component to perform its assigned roles and functions, and a formal characterization of the information that each actual and potentially available component can make available. For the purpose of this task, the important aspect of those characterizations is the conceptual model of the information, which must include some semantic characterization. If those formal models are available, it should be possible to automate the task of matching sources to requirements, using inferencing techniques or semantic analysis techniques. In many cases, however, human assistance may be required. This is an important research area.

## 3.5 Technical integration

When the functional roles have been assigned, and the data flows needed to enable and cause those functions to be performed have been identified, the technical integration activity is to define the detailed mechanisms by which those data flows will occur, using available technologies. This activity defines the detailed requirements for the system-specific actions of the components that will actually perform the data flow operations in the integrated system.

Technical integration is the task that turns an integratable component into an integrated component by ensuring that it is interoperable (see 3.3.2) with the other components with which it must interact.

This task has the following elements, depicted in Figure 8:

- Where necessary, specify the transformations and interface requirements for new or modified components to gather available data and derive the information needed for certain functions.
- From the interaction requirements, and the interface models of the components, specify the requirements for the interfaces to be used to activate functions and behaviors, and to obtain and deliver each set of information units.
- Identify the communications supported by the coordination and interface models of the components (see 4.4.3), and for each joint action, define the coordination model for the communications among the components involved in that joint action, including those that invoke the action and those that provide or acquire the needed data.
- For each interaction, from the coordination models and the interface models of the components involved, define the interfaces and protocols to be used to accomplish the communications needed for that interaction.
- Specify the technical transformations of the technologies, protocols, data structures and representations that are required to make the components involved in each such interaction interoperable, that is, to enable them to use the chosen interfaces and protocols for that interaction. (Most interactions involve a pair of components and at most one of them should need the assistance of a protocol or representation transformation.)
- Identify requirements for "bridges" and "wrappers" – intermediaries that convert technologies, protocols, data structures or representations, to enable communication between non-interoperable resources.

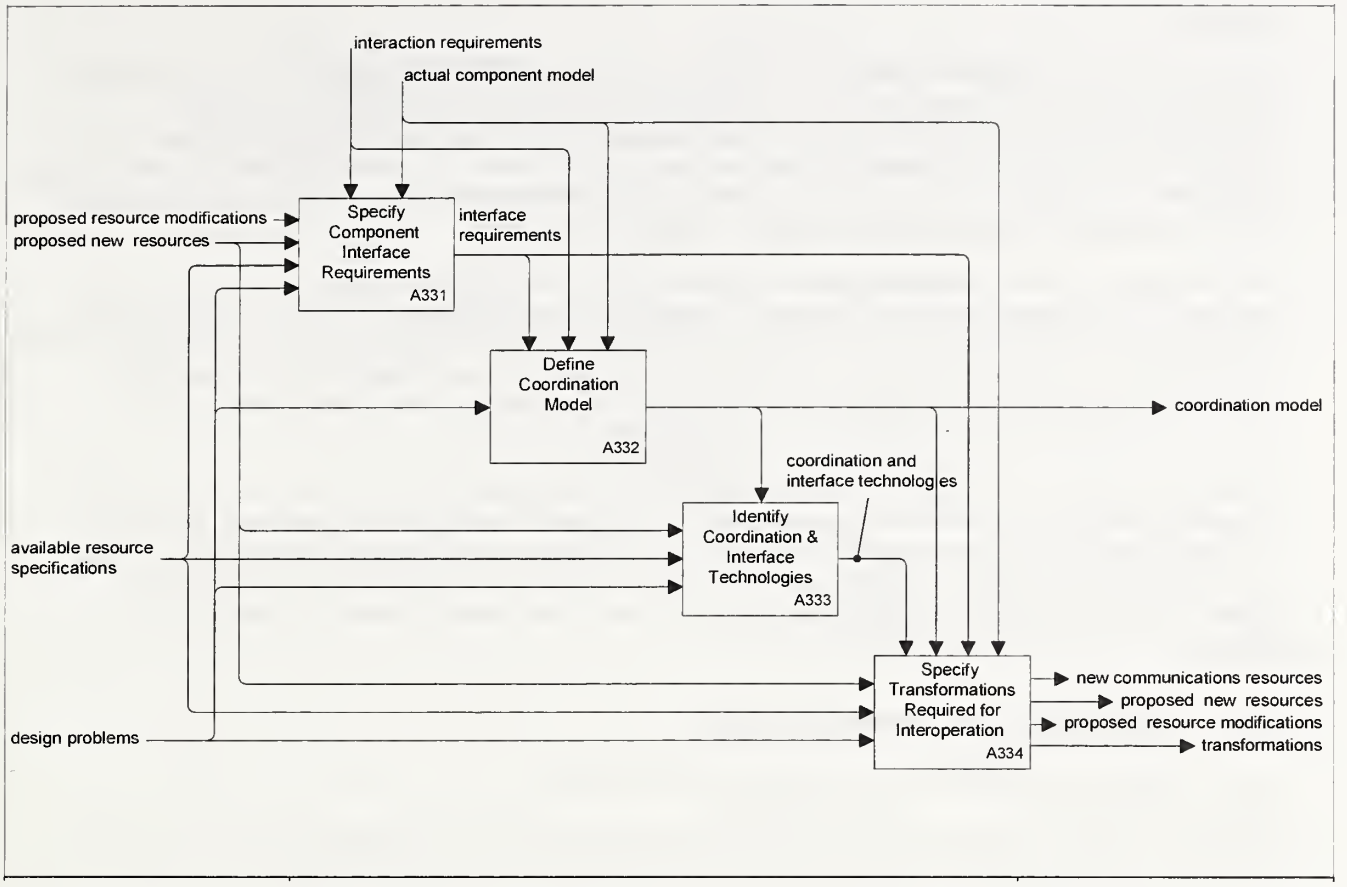


Figure 8 The Technical Integration Activity

Inputs:

- the actual components for the integrated system, together with their interface and coordination models, in some cases augmented by limitations on the scope and/or quality of the information at those interfaces, derived from further analysis and experience
- the interaction requirements for those components derived from the joint activities in the system processes, including those that enable or cause specific functions to be performed and those that provide needed information

Outputs:

- the coordination models for the interactions, and the specifications for the corresponding interfaces;
- specifications for additional component resources ("technical components"), whose sole function is to implement the communications and the protocol transformations
- specifications for resource modifications (usually "wrappers") to enable the components to exhibit the required interfaces
- specifications for specialized software modules that collect all the needed data, perform the derivations and transformations, and implement the flow of the information that fits the semantic model of, and interface form for, required inputs to certain components

Automation:

The specification of coordination models may be viewed as a planning problem as described in 6.7.1. The "initial task network" is the set of required interactions, derived from joint activities of the components, and the corresponding information flows. The "operators" are characterized by the interface descriptions, and their associated communications technologies, together with any available protocol converters and other "bridge" technologies. Certain details of the interface and communications technologies provide the "method templates" for building sequences of the operators that accomplish the required interactions.

Given sufficiently accurate formal specifications for the available interfaces to the components, and the corresponding data representations and protocols, it is possible to automate a process that reasons from the required information flows and the specifications for the available technical interfaces to the specifications for the data transformations. It is less often possible to do this for protocol transformations, although it may be possible to automate the specification of the control parameters for a general-purpose component that has been carefully engineered to perform the complex Protocol A to Protocol B transformation.

In those cases where the interacting components cannot use the same mechanisms (see 2.3) for the communications, it is probably not possible to automate the specification of a coordination model for their interaction. While it is in many cases possible to link components with different communication mechanisms, the transformation itself, and the determination of its validity, is usually highly dependent on the particular interaction to be automated. The engineering task involves understanding the desired behavior and the relationship between the behavior and the interface, which goes well beyond technical concerns.

## 3.6 Implement the integrated system

Acquire or build any additional functional and data resources required.

Configure or modify the selected resources to exhibit the behaviors required of the roles that have been assigned to them, including suppressing (or rendering harmless) the unwanted behaviors of partly integratable resources.

Configure, modify, or wrap the resources to implement the technical specifications for the communications.

Develop specialized software modules corresponding to the specifications for derivations and complex transformations.

Acquire or build the technical components (see 3.5).

Install, assemble and link the components.

Inputs:

- Actual component model of the system
- Specifications for new resources
- Specifications for resource modifications
- Specifications for wrappers and transformations
- Coordination, interface and protocol specifications for the interactions

Outputs:

- the integrated system

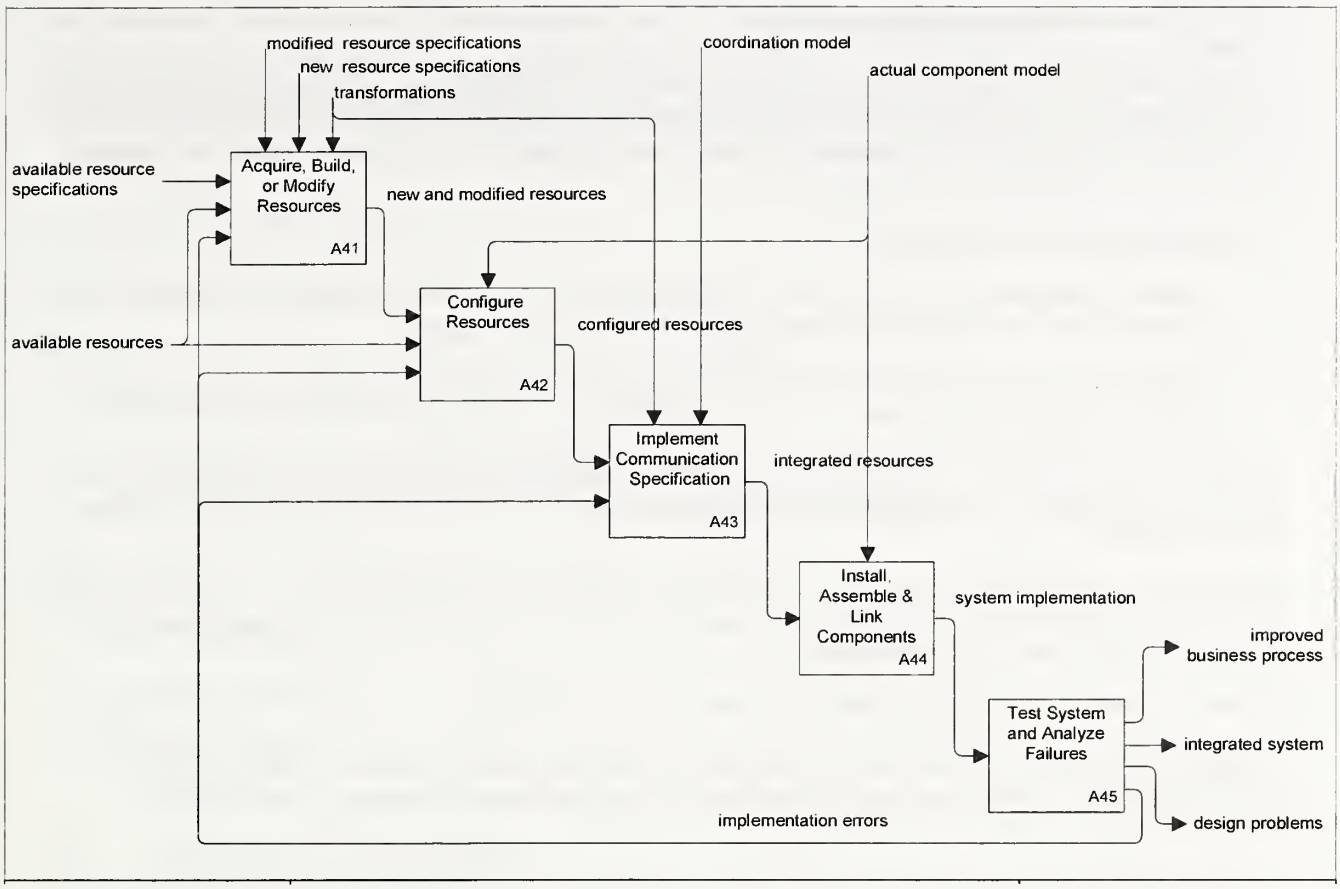


Figure 9 Implement and Test the System

Automation:

1. Selection of some of the technical specifications, and automatic configuration of the resources to use them, can sometimes be automated (e.g., via negotiation protocols, see 6.9). Other selections are perforce manual or built-in features of the resources.

2. Configuring and wrapping resources to accommodate terminology differences in the communications can be automated by any of several "terminology mapping" approaches, such as standard translation tables, common reference ontologies, "ontology mapping" techniques, and so forth, if the required knowledge base for the mapping is available (see 5.2.4 and 6.4).

3. Automatic construction of bridges may be possible, using a combination of planning and mapping technologies and a knowledge base of detailed technical specifications and conversion rules, as discussed in 3.5).

4. Automatic *configuration* of active resources to exhibit, suppress, or modify certain behaviors depends on built-in capabilities of the resource, but this could be an area of software engineering research or "best practices" work.

5. Automatic *modification* of active resources to exhibit, suppress or modify certain behaviors depends on exactly what kind of joint action is required, what behaviors of the resource are involved, and how the resource is constructed. Where the resource is itself a subsystem with an accessible component architecture (a white/clear box), this may be amenable to algorithmic approaches. (By definition, black boxes cannot be modified; they can only be wrapped.)

6. Resource discovery technologies (see 6.9.5), combined with formal description and a delivery or access protocol, may be used to locate and obtain additional required resources automatically.

## 3.7 Test and validate the system

In general, "testing and validation" is an appraisal of how well the system serves the intended function, as perceived by the systems engineer. What real effect the system has on the business process is a further evaluation that requires business metrics and subjective measurements that are beyond the scope of the information systems, and therefore out of scope of the integration process per se.

Testing and validation covers two processes:

- *validation* = determining that the behavior of a component implementation matches the specification for its role, i.e., that the subsystem produces the correct results for the required functions
- *verification* = determining whether the technical elements of the behavior of a component match the technical elements of its specified interfaces

*Validation* = formal proof, or some lesser evidence, that the component is integratable into its specified roles, and that the joint actions of the components produce the functions intended by the systems engineer. "Do the resources do the right thing?"

*Verification* = formal proof, or some lesser evidence, that the components are interoperable, i.e., the behaviors of the component are consistent with the technical specifications for their interoperability with other components in the system. "Does the resource do the thing right?"

*Testing* = the process of determining either of the above by experimentation.

Validation can be rigorous (with several different notions of rigor), or something simpler. It can be performed entirely by humans, or entirely automated, or performed by some human-assisted process, or not explicitly performed at all.

Unless the internal behavioral model of the component is completely exposed, formal validation is not possible. All that can be done is to produce experimental evidence that a given set of inputs (stimuli) consistently produces the expected measurable outputs (responses). An experimental validation process, however, is possible: if an exhaustive set of the expected inputs can be tested with all required environmental conditions and the results are those required for the specified functions, the component can be said to be *validated.* Where the set of possible inputs is infinite, or so large as to be intractable, a statistical sampling mechanism using input instances that provoke all characteristic behaviors may produce a credible validation. And, of course, any experimental mechanism is capable of *falsifying* the fulfillment of the role, by demonstrating production of incorrect responses to specific stimuli.

In a similar way, formal verification of interoperability is not usually possible, and a comprehensive test of technical responses to all valid stimuli is not usually possible, either. On the other hand, consistently correct technical behavior over a set of input instances that test all possible variations in response type, structure and representation, is usually deemed to be a verification, on the assumption that these technical behaviors of the component are invariant over the actual data they convey. Such tests are, of course, always capable of falsifying interoperability, by producing incorrect behaviors or technically incorrect results from valid inputs.

The problem with black-box testing is that the mapping between the external inputs and the internal behavior model of the component is unknown, as is the mapping between the internal behaviors and the measurable external results. Therefore, a comprehensive test of characteristic data for the external functional relationships may not actually test all characteristic internal behaviors! Even when an exhaustive set of inputs can be tested one by one, it is possible that the sequence of inputs affects the internal behavior in unexpected ways.

Automatic performance of an integration activity implies that the resulting interaction of the resources is *assumed to be valid* with respect to the intended function of the system, whether the automating agents explicitly validate it or not.

While validation is a very desirable engineering practice, it is not a requirement. There are tradeoffs between quick-and-dirty integrations that work for a specific set of applications, typically with later human intervention in the

process anyway, and carefully engineered integrations that will be expected to implement a critical business process without human intervention, and must be highly reliable. In the former case, the validation may consist of nothing more than running representative test cases and manually or automatically checking the results. But the latter clearly requires significant validation effort with some rigor, in order to reduce business risks to an acceptable level.

Inputs:

- the system implementation
- the target business process

Outputs:

- the integrated system, which enables the target business process
- feedback to the implementation processes on incorrect behaviors and failures
- feedback to the design process on the ability of the system to support the target business process

Automation:

- Automated testing
- Automated verification
- Automated validation

# 4 Aspects of integration

Because integration of components into a complex system can involve many concerns, it is useful to partition the space of concerns, for the purpose of identifying both problem areas and elements of solutions.

In this section we examine approaches to partitioning the problem.

## 4.1 Viewpoints and views

Viewpoint is a systems engineering concept that describes a partitioning of concerns in the characterization of a system. A *viewpoint* is an abstraction that yields a specification of the *whole* system restricted to a particular set of concerns [66]. Adoption of a viewpoint is useful in separating and resolving a set of engineering concerns involved in integrating a set of components into a system. Adopting a viewpoint makes certain aspects of the system "visible" and focuses attention on them, while making other aspects "invisible," so that issues in those aspects can be addressed separately. A good selection of viewpoints also partitions the design of the system into specific areas of expertise.

In any given viewpoint, it is possible to make a model of the system that contains only the objects that are visible from that viewpoint, but also captures all of the objects, relationships and constraints that are present in the system and relevant to that viewpoint. Such a model is said to be a *viewpoint model*, or a *view* of the system from that viewpoint.

A given view is a specification for the system at a particular *level of abstraction* from a given viewpoint. Different levels of abstraction contain different levels of detail. Higher-level views allow the engineer to fashion and comprehend the whole design and identify and resolve problems in the large. Lower-level views allow the engineer to concentrate on a part of the design and develop the detailed specifications.

In the system itself, however, all of the specifications appearing in the various viewpoint models must be addressed in the realized components of the system. And the specifications for any given component may be drawn from many different viewpoints. On the other hand, the specifications induced by the distribution of functions over specific components and component interactions will typically reflect a different partitioning of concerns than that reflected in the original viewpoints. Thus additional viewpoints, addressing the concerns of the individual components and the bottom-up synthesis of the system, may also be useful.

The purpose of viewpoints and views is to enable human engineers to comprehend very complex systems, and to organize the elements of the problem and the solution around domains of expertise. In the engineering of physically-intensive systems, viewpoints often correspond to capabilities and responsibilities within the engineering organization.

In this paper, we identify and use viewpoints to separate the concerns in an integration problem, primarily to distinguish the kinds of knowledge and engineering approaches that may be brought to bear to solve them. But any effort to automate the solution to a given integration problem must address all of the concerns in all of the viewpoints that impinge on that problem. It may do so by the divide-and-conquer approach used by human engineers, but it must ultimately address all of the concerns and integrate all of the solution elements derived from different viewpoint analyses into a cohesive solution.

## 4.2 The ISO Reference Model for Open Distributed Processing

The International Organization for Standardization (ISO) Reference Model for Open Distributed Processing (RMODP) [59] specifies a set of viewpoints for partitioning the design of a distributed software/hardware system. Since most integration problems arise in the design of such systems or in very analogous situations, these viewpoints may prove useful in separating integration concerns. The RMODP viewpoints are:

- the *enterprise* viewpoint, which is concerned with the purpose and behaviors of the system as it relates to the business objective and the business processes of the organization
- the *information* viewpoint, which is concerned with the nature of the information handled by the system and constraints on the use and interpretation of that information
- the *computational* viewpoint, which is concerned with the functional decomposition of the system into a set of components that exhibit specific behaviors and interact at interfaces
- the *engineering* viewpoint, which is concerned with the mechanisms and functions required to support the interactions of the computational components
- the *technology* viewpoint, which is concerned with the explicit choice of technologies for the implementation of the system, and particularly for the communications among the components

RMODP further defines a requirement for a design to contain specifications of consistency between viewpoints, including:

- the use of enterprise objects and processes in defining information units
- the use of enterprise objects and behaviors in specifying the behaviors of computational components, and use of the information units in defining computational interfaces
- the association of engineering choices with computational interfaces and behavior requirements
- the satisfaction of information, computational and engineering requirements in the chosen technologies

With respect to the activities described in Section 3, the definition of the business processes to be supported by the system and the specification of process roles and system roles are part of an enterprise view.

The assignment of resources to roles (3.3), however, may have aspects that are visible from each of the RMODP viewpoints. In particular, determining the *suitability* of a resource for a role requires a detailed analysis of the role requirements from the information viewpoint and the computational viewpoint and a corresponding analysis of the capabilities of the component in each of those viewpoints. That is, one must determine that the component is

capable of exhibiting the behaviors required for the role, which is part of its computational model, and that the component processes and produces all the information units associated with those behaviors. The identification of corresponding information units and their interpretations requires analysis of the information model for the component, and the association of those information units with particular behaviors is part of the computational model. Determining *interoperability* of the component with each of the components it must communicate with involves common choices for engineering mechanisms for the computational interfaces and common choices of implementation technology for the communications. That is, suitability issues are addressed in the enterprise, information, and computational viewpoints, while interoperability issues are addressed in the computational, engineering, and technology viewpoints.

In a similar way, the data integration approach described in 3.4 involves elements from all five RMODP viewpoints as well. In this approach, the relationship between the information model for the enhanced system and the information models of the components is the paramount concern — it controls the architecture of the solution. The computational models of the components yield the definitions of the required interfaces for the information flows. And the resolution of the interoperability problems between those interfaces (visible from the engineering and technical viewpoints) is the remainder of the integration design effort.

## 4.3 Categories of models

The traditional approach to systems specification is to develop multiple models of the system, each of which captures the relevant features of the system in some aspect and at some level of abstraction, resolve inconsistencies among the models, and assemble a specification for the system that consists primarily of those models. Formal languages and commonly used informal notations for expressing those models have developed over time, but largely in the absence of a formal discipline that would systematize their scopes and viewpoints. Moreover, the formal specifications for "legacy" subsystems and externally supplied (reusable) components often take the form of such models. There are two general categories of model: conceptual models and technical models.

### 4.3.1 Conceptual models

*Conceptual* models capture descriptions of a system in terms of the business objects, the business processes, the associated computational functions and the associated information units. Conceptual models are intended to be technology-independent. They may explicitly represent many of the constraints associated with the modeled concepts in a formal form. Conceptual models can be associated with different levels of abstraction and varying degrees of detail, and mappings between conceptual models at different levels of abstraction typically refine one object into multiple objects and one function into multiple component functions.

### 4.3.2 Technical models

A *technical* model of a system element is a specification for that element in terms of the specific information technologies used to implement the system. It embodies engineering decisions that relate to the technologies and the specifics of the implementation itself. In particular, a technical model captures the mechanisms (to be) used and may capture the details of their use. The technical model may undergo further transformations in becoming an implementation, but these transformations tend to be well-defined rote mappings (e.g., the transformation of Java [110] to the interpretable byte-code for the Java Virtual Machine).

### 4.3.3 Model mappings

At a sufficiently low level of abstraction, there may be a regular mapping from modeling language constructs to implementation constructs. Such a mapping allows the conceptual model to specify a particular technical solution (implementation) for storing data, exchanging data, performing standard data manipulations and computations, and implementing function sequences. That is, the conceptual model can be translated directly to the data structuring language for a database (e.g., SQL [60]), or the interface language for a communications mechanism (e.g., IDL or WSDL, see 6.3.2), or the object models and invocation paraforms for a programming language (e.g., Java or Visual

Basic). While these implementation models preserve many of the characteristics of the conceptual model, the transformation will be colored by the inherent mechanisms of the technology used. This will add additional behavioral constraints and representation choices.[2]

## 4.4 Model domains

In addition to separating levels of abstraction, models can be characterized by the viewpoint on the system or component that defines the domain of system description they are attempting to capture. Unfortunately, the traditional kinds of models that systems engineers make of their systems don't all fit neatly into the RMODP viewpoints. In this section, we attempt to identify the principal "domains" of models. In general, these are collections of models that tend to overlap in the concerns they capture. These models also often represent a continuum of concerns from a conceptual level down to a detailed technical level with no clear break.

### 4.4.1 Information domain

Models in the information domain include "semantic models," "information models," and "data models." All of these models are views of the system from the information viewpoint.

An *information model*[3] is a model of the business objects, the relationships among them, the properties of these objects and relationships, and the information units that describe those properties, to the extent that any of these are important to the processes of the system. In particular, for a distributed system, it is useful to define the *shared information model*, that is, the objects, relationships, properties and information units that are provided and used by more than one component in the system. Among these will be all of the objects and information that are important to the joint actions. Information models are conceptual models – their description of objects and information units is independent of the mechanisms chosen for their capture, transfer and storage.

Most information models declare objects and information units formally, but they *define* them using natural language text. *Semantic models*[4] are conceptual models that formally define the meaning of the objects and information units, in terms of fundamental notions, properties and behaviors. Semantic models define the meanings of the terms used, by reference to taxonomies, thesauri, formal dictionaries or axiomatic characterizations. These models are intended for use with corresponding artificial intelligence methods. Such models are sometimes called *ontologies*.

*Data models* are technical models of information — they define the information units in terms of their organization and representation for capture, transfer and storage. Data models include database models, data exchange languages like ASN.1 [119] and XML [120], and the data structures of programming languages. Programming and database models generally specify the logical organization and representation types for the data, but not the physical representation of the data, which is encapsulated in the program/database implementation. But in a distributed system it is necessary to specify the physical organization and representation of the information as it will appear in the flow between components.

---

[2] The mapping of concepts from the conceptual language to structures in the implementation language is usually many-to-1. It is often possible to choose a "reverse mapping" that produces a model in the conceptual language from the formal description of a given implementation. But the resulting model is at the same level of abstraction as the implementation itself and reflects all the engineering choices made. This means that it is unlikely that a round-trip mapping from conceptual model to technical model and back would result in equivalent models.

[3] The terms *information model* and *data model* have traditionally been used to refer to models that include some conceptual and some technical elements. The terms have also been used to distinguish the emphasis on the conceptual elements from the emphasis on the technical elements.

[4] The term *semantic (data, information) model* was often used in the 1980s to refer to information models. Similarly, the term *semantic object model* was sometimes used in the early 1990s to describe interface models that were independent of protocol (see 4.4.3). But in the late 1990s the terms *semantic model* and *ontology* came to mean models of the meanings of terms.

There are two levels of mapping that occur in models in the information domain, both of which are loosely termed "representation." Conceptual objects, information units and their relationships are mapped to (represented by) data elements and structures in specifying interfaces. Data elements and structures are mapped to (represented by) sequences of bits and character codes in the actual data exchanges. The former is a set of engineering decisions; the latter is a rote mapping associated with the implementation technology.

### 4.4.2 Functional domain

Models in the functional domain describe what systems and components do.

Behavior is what the system/component does; function is the result that behavior achieves (see 2.2). Behavior is an inherent characteristic of the system/component; function is an exploitation of inherent characteristics. A resource can have a built-in behavior that supports several functions of a larger system without being explicitly designed to support those functions. The behavior of a resource can be influenced by interactions, environmental conditions, past actions, and configuration.[5]

Since most behaviors are motivated by some function intended by the designer, behavioral models often contain references to the target function. But a behavioral modeling technique usually describes behaviors without reference to function.

One kind of behavioral model, sometimes called a *state-space model*, models the behavior of a system as the effects the system activity has on the observed external environment and on its internal environment. That set of effects can be seen as implementing more than one function, if different subsets of those effects are considered useful with respect to some target business process.

Another kind of behavioral model, sometimes called an *execution* model, formally models the actions of the system in terms of a set of elementary actions of some abstract machine over a set of time frames. When the actions are required to be sequential, the model is said to be *procedural*; when the actions are specified as responses to stimuli, the model is said to be *event-driven*. And it is in some cases possible to prove that the actions so modeled accomplish a given function.

A *computer program* is in a sense the "lowest level" form of execution model — the machine is actual, and the behavior model in terms of its elementary operations is the directions to perform those operations.

A true *functional model* reflects a particular viewpoint on a behavioral model — it models behavior as it relates to a particular goal. A description of a function might describe a desired behavior, or the exploitation of an inherent behavior, or the desired outcome of an unspecified behavior. Importantly, a functional model of a component or system ignores those elements of its behavior (the state changes or actions) that are irrelevant to the goal(s) in question. The functional model is therefore an *incomplete* model of the behaviors of the system.

A *process model* is the "decomposition" of a particular function: it is a logical or temporal graph of activities (behaviors) of component resources whose end result is the intended effect on the observed environment. It describes *how* the system accomplishes the function. It resembles an execution model, but the elementary actions in it are required behaviors of the components. In most cases, the activities in the process represent exploitations of inherent behaviors of the components. And all of the steps in the process are traceable to the required end results. The process model ignores any ancillary behaviors of the components that may be concurrent with, part of, or consequences of, the desired behaviors.

Whether models in the functional domain are conceptual or technical is mostly a matter of the level of abstraction. As the virtual machines that underlie programming languages become more powerful, the elementary operations that

[5] "Configuration" can refer to a design activity, which permanently alters the behaviors of the component to perform certain functions. "Configuration" can also refer to a deployment activity in which the designed-in behaviors of the component are refined by setting controls and parameters that were designed to permit these refinements. In the most flexible situations, these parameters can be altered during operation of the system. It is in the deployment sense that we use "configuration" here.

have direct implementation also become more powerful, so that what was an abstraction that required a further engineering choice and detailed coding in 1980 is now a pre-defined operation in a Java or Perl environment. So the point at which an execution model becomes an implementation is not fixed. But at the same time, elements of the "virtual machine" that are assumed by the component builder – engineering choices made by the machine designer – may be important technical elements in the integration problem.

Similarly, the assignment of models in the functional domain to RMODP viewpoints is difficult. Process models are views in the enterprise viewpoint, but function and behavior models may appear in the enterprise viewpoint, the computational viewpoint, the engineering viewpoint or even the technology viewpoint, depending on the focus and level of abstraction. This should not be surprising — models of what the system/component does and how it does it pervade the concept "specification of the system."

### 4.4.3 Interface (coordination) domain

In the RMODP computational viewpoint, a component is defined by a set of behaviors associated with interfaces and two specifications for the use of those interfaces: one derived from the component designer's concept of the functions the component is intended to support, and one derived from the system designer's intent to exploit the behaviors of the component in accomplishing the functions of the system.

*Interface* models specify the mechanism that a component exposes to support interactions, including the nature of the control flow, the nature of the information flow (e.g., database, message, blackboard, operation invocation), the details of the unit communications (including their data models), and the rules for their sequencing. In general, interface models are stated as a set of specifications for the responsibilities of the component that provides that interface in the interaction (e.g., the server, the file writer, the data repository) and the responsibilities of the other parties (as client or reader) are implied. That is, the interface model specifies the view of communications as offered by the resource.

*Coordination models* specify the detailed interactions between the components, and the related control and information flows, that enable the activities in the system process model to occur. They describe the interactions that occur as part of the execution of the process (performance of the joint actions) in terms of the interfaces exposed by the interacting components.

*Protocols* are rules for the implementation and sequencing of unit communications for a class of interfaces — they specify a set of rules that guarantee interoperability among implementations of some mechanism. A protocol can usually be applied to any interface model that uses the selected mechanism, but some details of interface specifications may be protocol-specific. A protocol defines the mapping from the interface specifications to the physical representation of the communication message units, and the rules for formation, transmission, receipt, acknowledgement and sequencing of message units, without regard to their content. While the term more narrowly applies to communication networks, the same concept applies to file systems and databases. Examples are: Hypertext Transfer Protocol (HTTP) [121], Simple Object Access Protocol (SOAP) [123], Simple Mail Transfer Protocol (SMTP) [122], and SQL Call-Level Interface [124]. Unless there is agreement on protocol, it is physically impossible for a communication between two components to take place.

*Presentation models* are a specialization of interface models for the case where the component or the interaction mechanism involves a human being, that is, where the flow itself is enabled by, and possibly implemented by, actions of a human agent.

All interface models, presentation models and coordination models are essentially technical (see 6.3.2). While it is possible to talk about a "conceptual interface," interface modeling languages rarely support that. They inevitably make assumptions about the choice of mechanism (database, message, operation invocation, blackboard) and often the choice of protocol. Those that purport to be "abstract" choose a mechanism and describe interfaces for that mechanism, but avoid protocol-specific elements. For example, the OMG Model-Driven Architecture [118] assumes that the mechanism is operation invocation and defines interfaces accordingly, but it avoids choosing among the common operation/webservice invocation protocols (CORBA [84], Java [127], SOAP [123], .Net [109]), and explicitly identifies those choices and the related protocol-specific interface elements to be a lower-level engineering decision ("platform-specific models"). Similarly, the Webservice Definition Language (WSDL) [79]

permits definition of an interface that is abstracted over operation invocation and messaging. It provides, but does not require the use of, mechanism-specific elements for more detailed specifications using those mechanisms. But in both cases, the assumption that the interface mechanism is a *direct* communications mechanism makes it difficult to map the interfaces so specified to a publish-and-subscribe blackboard protocol or to interchanges via shared databases.

For similar reasons, the distinction between the computational views of the system and the related engineering views can be fuzzy. But for integration purposes, there are three distinct views:

- the abstract computational view – the component provides the needed function or information and an interface to it
- the engineering view — the interface uses a particular mechanism and is formally modeled in that way
- the technology view – the interface uses a particular protocol

And each of these views defines a different subproblem that requires different tools to solve.

### 4.4.4 System domain

*System structure* models describe the organization of the resources in the system, including the roles of components within that system, and their corresponding communication and control relationships. High-level models of this kind are enterprise views of the system. More detailed system structure models are engineering views of the system.

*Policy models* are conceptual models that specify business rules for the conduct of business operations, and in particular for the reliability of systems, the timeliness of activities and information, and the accuracy and security of information. They are enterprise views in RMODP terms.

There are corresponding engineering models for implementing policy models. They typically give rise to additional components in the system structure models, to constraints on coordination models, and to additional protocols and elements of protocols.

Detailed system structure models may include identification of backup file systems, redundant processing systems, and backup or redundant communication systems, to ensure availability, reliability, failure protection, and recovery. They may also specify proxies and reflectors and other components designed to facilitate timeliness. That is, these engineering views of system structure are designs for the support of the enterprise views of the system, including its policy models.

*Network models* are a kind of system structure model that specifies the logical and/or physical structure of system communications, including the logical connections and intermediaries, and the physical links and communication appliances.

## 4.5 Integration aspects derived from model domains

An integration aspect is a set of concerns for the system engineer about the compatibility of components with their roles or with their partners in a given set of interactions.

Because the existing documentation for reusable components is usually in the form of models of the kinds identified in Section 4.3 above, those models, together with text documents and observations, form the basis of the system engineer's knowledge of those components. And in any effort to automate the related integration tasks, those models will form the basis of the tool's characterizations of those components.

On the other hand, the actual integration processes described in Section 3 rely heavily on the ability of the system engineer to separate conceptual concerns from technical ones, in order to identify problems (and non-problems) correctly and take the right kinds of engineering actions in resolving them. So in those cases where the models do not clearly separate technical and conceptual concerns, it is necessary for us to do so.

In this section, we will identify integration aspects based on the kinds of models of systems and components that are commonly made and the conceptual vs. technical distinctions in concerns that the system engineer must make when

using those models. As observed above, these model domains and levels of abstraction do not always clearly conform to RMODP viewpoints. And in the same way, the integration aspects herein defined will not necessarily conform to RMODP viewpoints.

A further distinction for the purposes of automating integration activities is the extent to which the engineering process can be automated or supported by automation. Almost all purely technical problems can be solved by automated tools, if some necessary transformation components are available. And in general, conceptual problems cannot be solved by automata. It turns out that the degree to which the engineering activities related to conceptual problems can be supported by automation corresponds fairly well to the model domain!

This analysis of the relationship of concerns for automating integration activities to concerns for categories of model leads to the definition of five integration aspects described below.

### 4.5.1 Technical aspects

The *technical aspects* correspond to the engineering and technology viewpoints in RMODP. They relate to the resolution of implementation issues in communication, data and control flow, such as the selection of underlying communications protocols, message structures and representation of content. There are two principal technical aspects: coordination and data.

What is required in the *coordination integration aspect* is that there is agreement between components on the coordination mechanisms, interfaces and protocols for their interactions. Without agreements on these, or intervention to bridge the differences, communication between components is not possible.

What is required in the *data integration aspect* is that components agree on the organization, structure and representation of information in the interactions between them. These concerns are sometimes part of, and sometimes in addition to, the agreement on the interfaces. That is, some interfaces provide for the exchange of variable data sets that are defined by separate agreements. In fact, the effort to standardize interfaces for various kinds of tools has led primarily to standard transaction definitions that do not specify the details of the information that accompanies a transaction, and those detailed information specifications become many separate standards and agreements. The data integration aspect deals with identifying common information specifications or resolving differences between alternatives.

### 4.5.2 Semantic aspect

The *semantic* aspect of integration concerns agreement of communicating parties on all the common concepts related to their interactions and on the terms used to refer to those concepts. What is required in the semantic aspect is that the interpretation of information units transferred between components is consistent relative to the business processes.

### 4.5.3 Functional aspect

The *functional* aspect of integration concerns behaviors of the components, with respect to their roles in the system, and with respect to each other's expectations.

What is required in the *functional integration aspect* is

- that the functional and behavioral characteristics of components are consistent with their roles in the business process activities; and
- that the resources agree on the objects to be acted on and the actions to be taken for each communication, to the extent that the effects of the actions are related to the goals of the interaction and the overall functionality of the system.

The functional integration aspect, therefore, deals with the suitability of components for their roles (see 3.3.1).

### 4.5.4 Policy aspect

The *policy integration aspect* is concerned with the ability of the system as a whole to support the business processes well. What is required in the policy aspect is that the behaviors of components and their interactions are consistent with the policies of the enterprise in areas that impose requirements that are largely separate from the functions performed. These include security, reliability, availability, accuracy, and timeliness.

Most of the concerns in the policy aspect relate to the technical considerations for implementing the policies.

### 4.5.5 Logistical aspect

The *logistical* aspect of integration deals with impacts of the integrated system on the business itself. Most of the concerns in the logistical aspect relate to engineering considerations that affect technical decisions in less predictable ways. They relate to tradeoffs between limitations on the integration process itself and overall value of the resulting system, e.g., cost, time, and resources for the integration process, and similar considerations for the resulting system: lifetime costs, resource consumption, impacts on other activities of the business, and future plans.

# 5 Integration concerns

This section discusses issues to be addressed in integrating components into roles in larger systems. These include impediments to integration and interoperability in existing components and other concerns for the specification of roles and contracts.

We have found it useful to collect these issues into five categories, which correspond roughly to viewpoints discussed in Section 4. In addition to appearing in the same engineering viewpoint, the problems in each category tend to be similar in the way in which automation can support their solution, and in the degree to which the solution can be automated.

The simultaneous resolution of the semantic, functional and technical problems is critical to enabling the functions of the integrated system, and the reconciliation in any given aspect is typically dependent on the reconciliation in all of the others. The resolution of the policy and logistical concerns is critical to getting value from the integrated system.

## 5.1 Technical concerns

Technical impediments to interoperability are mismatches in the details of behavior and information representation *at the interface*, including protocols used, communications technologies, data structure and representation, and expected quality-of-service characteristics. That is, technical impediments are *interoperability* problems, in one way or another, and most interoperability problems are technical impediments.

In general, technical conflicts lend themselves most to automated identification and resolution. Several recently developed tools and techniques (see 6.5.1 and 6.8) are aimed at the automatic provision of technical components for the solution to certain kinds of technical conflicts. In general, however, such tools depend on human engineers to identify and characterize the technical conflict.

### 5.1.1 Connection conflicts

A *connection conflict* arises when the two resources disagree on the technical specifications for performing the communication.

The technical specifications for communication with a given component are contained in its interface specifications (see 4.4.3). The interfaces in turn identify the communications mechanisms and protocols that the component expects to use.

When the interfaces exposed by the communicating resources do not use the same mechanism, the interfaces are said to have a *communication mechanism conflict*.

Even when the two interfaces use the same general mechanism, the specific protocols they use may be different. Further, remote communication mechanisms in the "application layer" of the Open Systems Interconnection Reference model [116] require the use of "lower-layer protocols" to permit the application layer protocol to function at all. If the two interfaces match in mechanism, but do not match in the application layer protocol and in all supporting protocols, the interfaces are said to have a *protocol conflict*.

Either of these mismatches can be accompanied by control conflicts (see 5.1.3), because many of the protocols assign asymmetric roles to the participating resources.

Examples:

- One component expects to receive data through a (Unix) "pipeline," but we need to integrate it with a component that only reads and writes to files.
- The client resource uses the Java Database Connectivity (JDBC) protocol, but the database resource is a relational database expecting SQL.
- A procedure-call client resource uses the Distributed Component Object Model (DCOM) protocols but the server is built with CORBA.
- A software component must provide data to an application whose only data entry interface is a graphical user interface (GUI) intended for human input.

Approaches to solution:

When all that is required to enable interoperation is to modify the mechanism or protocol of the information flow, and/or its underlying protocols, without reorganizing the content of the communication, the integration solution is referred to as a *transformation on the connectors* [51].

For example, it is possible to resolve the pipes-versus-files conflict using a scripting language. Output redirection can be used as a pipe-to-file adapter, and any program that can copy a file to standard output can be used as a file-to-pipe adapter.

But many transforms between mechanisms require reorganization of the information content and may well be steered by the information content itself. The components that implement these more complex data-driven transforms are commonly referred to as *wrappers*. In many cases, even when the wrapper is only performing a protocol transformation, the two protocols do not factor the operations identically (see also 5.3.1). In such a case, the wrapper must perform a many-to-one or one-to-many conversion between the communication units of the protocols, and in the worst case, a many-to-many conversion.

In some cases, it is necessary to modify the component itself to give it a new interface, because no transformation of the existing connector is feasible. That is, the existing connector does not provide sufficient information at the times needed for any wrapper to perform the transformation properly. Implementation of the new protocol requires access to internal information in the resource.

### 5.1.2 Syntactic conflicts

A *syntactic* conflict occurs when the communicating components use different data structures and representations for identical concepts. The concept in question can be a type of object, action, property or relationship, or an instance of any of them. The conflict can be as simple as a disagreement on the spelling of a name, or as complex as a different data organization and a different approach to data representation.

Examples:

- One system uses ASN.1 to represent the data schema and the Basic Encoding Rules to represent the corresponding data; the other component uses XML Schema to represent the data schema and corresponding XML representations of the data.

- One system uses ISO 10303 [19] specifications to represent a Bill of Materials (BOM); the other system uses the Open Applications Group Integration Specification (OAGIS) [125] "Show BOM" transaction.
- One component models its interfaces in Java and uses the Java Remote Message Interface representation for the messages; the other component models its interfaces in the ISO Interface Definition Language and uses the SOAP representation for the messages.

Approaches to solution:

In general, what is required is an intermediary component that translates between the two representations. And the intermediary will be specific to that set of interface transactions. In many cases, the intermediary will perform both a protocol transformation (see 5.1.1) and a syntax translation.

In some cases, the intermediary can be constructed in a 3-step process:
- Use a language-specific (but model-independent) translator to convert the source data schema or interface model to an equivalent model in the language of the target model. The output of this tool is the equivalent model in the target language and the implementation software to convert the corresponding data elements to target data representation conventions.
- Using a language appropriate to the target representation conventions (such as Express-X [21] or XSLT [78]), specify the model-specific conversions (name translations and data reorganizations) from this equivalent representation of the source to the target representation, and use a tool for this language to generate the corresponding data mapping implementation.
- Build the intermediary by coupling the two generated translators and providing the data routing. (This can be a difficult task, depending on the protocols involved.)

When the interaction is more complex than one-way or request and acknowledge, the matching conversion for the flows in the reverse direction must also be constructed.

### 5.1.3 Control conflicts

A *control conflict* occurs when the components embody incompatible assumptions about the flow of control in their interaction. This is a *technical* problem when the two components agree otherwise on their functional roles: who is providing what information to whom and who is performing which function.

Examples:
- "too many leaders" – Both components expect to be the "client" component, invoking a service (operation, procedure) provided by the other component; neither expects to be a "server" and respond to the other's requests.
- "no leader" – Both components expect to be the server component, providing operations to be invoked by another component; neither expects to be a client and initiate a request.
- A "publisher" eomponent expects to be called by a "subscriber," who registers interest in a certain set of information and provides a "call-back" entry point, and then to provide notifications of changes in that information by invoking that call-back; the "subscriber" component expects to call the publisher and register interest in a certain set of information, and to call the publisher back later to ask for any changes in the information in which it has registered interest. In this case, there is a nominal match on the subscriber/publisher model, but all the subsequent interactions have too many leaders.

In the general case, a control conflict is a conflict between "push" and "pull" models of some part of the interaction protocol. That is, a *push client* component expects to provide information at a time of its choice by initiating a communication that "pushes" the data to a *push server* component, who is expected to accept the data when it is sent. A *pull client* component expects to retrieve information at a time of its choice by initiating a communication to "pull" the data from a *pull server* component, who is expected to respond. The "too many leaders" problem arises when a push client must transmit information to a pull client; the "no leader" problem arises when a pull server must transmit information to a push server. And these *protocol roles* can change for different elements of the overall interaction between the components, as in the publish/subscribe example.

Approaches to solution:

In the "too many leaders" case, one of the clients must be converted into a server. This can be more or less difficult depending on the architecture of the component. In most cases, the integrator must construct an intermediary component that is both a push server, who accepts and queues messages from the push client, and a pull server, who delivers them to the pull client on request.

The "no leader" case is easier. The integrator supplies the component that implements the exchange by being a pull client to the pull server and a push client to the push server.

Either of these cases can be further complicated by a protocol mismatch that involves a many-to-one or many-to-many transformation between the sets of protocol messages that must be exchanged (see 5.1.1).

### 5.1.4 Quality-of-service conflicts

A *quality-of-service conflict* occurs when the behavior of a component does not always satisfy technical requirements derived from "policy" concerns (see 4.5.4). Such requirements relate to performance, reliability, environment of use, and similar considerations (see also 5.4).

Quality-of-service conflicts can also occur when one component is built to satisfy such requirements and *expects* components it communicates with to satisfy those requirements as well, even though the system does not have those requirements. That is, quality-of-service conflicts can arise because the system role creates a requirement for technical behaviors that the component does not satisfy, or because another component that must interoperate with this component *expects* this component to satisfy such a requirement.

Examples:
- A client component expects computed results to be accurate to 10 decimal places, but the server performs the computations using "single-precision" machine arithmetic (7-8 decimal places).
- A component is expected to operate in a real-time system and respond within certain time constraints.
- A component is expected to use secure communications mechanisms for certain kinds of transactions.
- A component is expected to restart automatically after a failure, and possibly to recover its internal state or to negotiate the communications state at restart, in order to support fault-tolerance.

If the quality-of-service requirement in question is also a requirement of the integrated system, then the problem is that one of the components fails to meet that requirement, and is at best partly integratable into that role. That is, the selected resource might not be integratable into that role at all, for reasons that relate to the quality aspects of the system (see 3.3.1).

If the quality-of-service requirement is not a requirement of the integrated system, then the problem is to eliminate any undesirable behaviors of the component that expects that quality-of-service when its expectations are not met. Such a component may detect *and respond to* a quality-of-service-related condition (timing, security, redundancy, etc.) that is irrelevant in the integrated system context, in such a way that the response interferes with the intended interaction between the components.

Approaches to solution:

True quality-of-service requirements for the role are usually very hard to retrofit into under-designed resources. In general, the component must be modified or replaced. Some optimistic "solutions" can be used, especially when the failure to meet the requirements is rare in actual service, such as:
- The component can be left sufficiently under-utilized that deadlines will be met.
- The component can be wrapped with a security preprocessor that validates or rejects every transaction, as long as that validation does not require data internal to the component.
- The component can be wrapped with a script that restarts it if it crashes, but state recovery will be at best approximate.

By comparison, undesirable side-effects of unnecessary quality-of-service expectations are often easy to eliminate. The requirements in our examples can be eliminated as follows without modifying the components:
- Real-time deadlines and timeouts can all be set to very lax values.

- Security classifications can all be set to the most public and unprotected policy.
- Fail-soft recovery messages can be turned off or intercepted by an intermediary that executes the proper protocol for a notification recipient and otherwise ignores them.

### 5.1.5 Data consistency conflicts

An interaction between components may rely on information that is available to both and not directly communicated in their interaction. When that information is acquired from different sources, acquired at different times, or deduced from different underlying data, the two communicating components may have inconsistent versions of that information. A *data consistency conflict* arises when such an inconsistency occurs and affects the behavior of the components in a way that interferes with their joint action or the overall functions of the system.

Examples:

- The Enterprise Resource Planning (ERP) system initiates a materials order based on an internal entry made from a supplier's catalog entry, but the procurement system is unable to place the order, because the supplier's online catalog no longer contains that item (as such).
- The manufacturing scheduler asks the database system for the location of the materials container for lot 604, finds that it is still in an in-buffer for an automatic storage and retrieval system (ASRS), and sends a command to the ASRS to cancel the "store" command for that container, but the ASRS controller reports that the command has already completed — the ASRS database transaction to update the location occurred between the two actions of the scheduler.
- The ERP system releases an assembly lot that requires 10 widget parts, but the materials preparation station is unable to setup the lot, because it only finds 8 widgets in the bin — the ERP system is calculating "widgets available" as the difference between widgets received and widgets drawn, but yesterday's widget shipment is still in the packing crate; it will be available to the materials prep station in 2 hours (but by then the first processing station will be busy with the next job).

Approaches to solution:

In many cases, the problem is that there are two sources for the information that are not adequately aligned. For example, one of the components may be maintaining a private file or database for that information that is not being regularly updated from the "primary source" — the component whose responsibility is to maintain that information. This is an oversight in the design of the integrated system that must be rectified by adding to the relevant process specifications the additional joint action for the data set updates. (In some cases, that data set cannot be updated while the component is running, or won't be re-examined after the component startup, and that may require modification of the component itself.)

When the problem is access to a common source but at different times, the data may be modified in the interim, so the two components get different values for the same nominal information unit. This is technically termed a "race condition" — the result depends on who gets there first — and it can be a very complex problem. What is happening is conceptually a joint "transaction" on this information set, involving two readers, that is not implemented as an "atomic" transaction, and probably cannot easily be. External controls must be imposed on all the joint actions that are involved in this data flow, including the update, in order to get consistency where needed. In some cases, this is best solved by scheduling or synchronizing joint actions that are not otherwise related. In some cases, it may be sufficient to force one of the components to re-fetch the information at the proper time, or conversely to prevent one of the components from acquiring a new version of that information while it is involved in the joint action. Or it may be possible to add that information unit to the direct communications for the joint action. Or it may be possible to for one component to recognize the inconsistency as an exception state and take an appropriate action (as in the ASRS example). Every race condition problem is unique and every solution is specific to the case at hand.

When the problem is inconsistent derivations, it is usually the case that one component has the information unit verbatim in its internal model, while the other component is deriving it from the information in its internal model. When the derivation is simply wrong, there is no solution other than to modify the affected component. In other cases, the derivation is correct, but there is a (formerly unrecognized) requirement for consistency between the

information units on which the derivation is based and the information unit the other component is using. And in those cases, the underlying problem is either multiple sources or "atomicity" as described above.

Unlike most technical conflicts, solutions to data consistency conflicts will not soon be automated, simply because much detailed analysis is required to determine exactly what the problem *is*, and what approach to solution will work for the system in question.

## 5.2 Semantic conflicts

The semantic requirement for integration is that communicating parties have common concepts and agree on the terms used to refer to those concepts. Resources must agree on the objects, properties and actions that relate to their roles in the larger system and the relevant relationships among those.

The only semantics that need to be preserved at a communication interface are the semantics that the communicating resources need to share in order to have that interface produce the expected joint action. In general, this means:

- The resource and the system must agree on the "sense" — the models of objects and actions and their properties that are important to the role of the resource as perceived by the system.
- The communicating resources must agree on the "reference" — the names for those objects, actions and properties that appear at the interface.

*Semantic conflicts* result from mismatches between communicating components in the conceptual model of the data and activity spaces in which they must interact, or in the interpretation of references to object types and instances in that space.

There are two fundamentally different kinds of communications:

- ***direct*** = where the speaker is aware of the specific listener(s) and the purpose is to produce a set of specific behaviors on the part of the listener(s), or to report the status or result of a requested behavior
- ***indirect*** = where the speaker is unaware of the specific listener(s) and the purpose of the speaker is to provide information that will be used in achieving some behaviors of other parties known to the system engineer

And there are two corresponding semantic issues:

- When a client communicates directly with a server, the server must interpret the communication appropriately to execute the action/behavior expected by the client. When two resources are actively collaborating, each must interpret the communications of the other appropriately and provide the behaviors expected by the other that are relevant to the joint action.

- When a subsystem communicates indirectly, *any* recipient must interpret the information provided appropriately to execute some action/behavior expected *by the systems engineer*. In general, the *speaker* has at most generalized expectations, for that behavior.

When a component communicates indirectly, all of its utterances are (or should be) statements of fact — facts, as the speaker understands them, that will be of use to other resources. The facts need only be those that are sufficient to convey the intended information, that is, the facts provided may have implications that are an expected part of the interpretation of the utterances. The recipient, however, may or may not make such expected inferences, and may make additional inferences, depending on its intended use of the information. For example, a relationship between objects A and B might be represented by a pointer from A to B; it is not necessary for the communication to contain the "reverse" pointer from B to A, even when the relationship from B to A is critical to understanding the communication – the one pointer implies (represents) the relationship.

By comparison, when a component communicates directly with another component, only some of its utterances will be statements of fact. Some will be commands that elicit specific behaviors; others will be requests for information, acknowledgements of receipt of commands or information, and other utterances that arise from the protocol itself rather than the interaction requirements.

Note — The information sets exchanged indirectly by manufacturing software systems are typically large complexes that must be interpreted by many different systems with different behaviors. As a consequence, the "meaning" of the elements of the data set, with respect to a large set of potential recipient behaviors, is important. The problem is: does the recipient understand the producer's description of the product/process well enough to determine the corresponding characterization of the product/process in the "world model" of the recipient, where the producer's description and the recipient's characterization can both be very complex.

In general, the question of whether the identification and resolution of semantic conflicts can be automated is open! Several techniques for the formal representation of semantics of various kinds have been developed (see 6.4.1), and other technology has been developed to make semantic inferences from such representations and determine semantic equivalences (see 6.5). Some of these approaches may serve to automate the resolution of some semantic conflicts.

### 5.2.1 Conceptualization conflicts

A *conceptualization conflict* occurs when communicating components have interface models representing incompatible factorings of the same conceptual domain. The conceptual representation of the problem space is a view of the space that is typically closely aligned with the algorithms used by that component to perform its assigned functions. And its interface models will normally reflect that internal view of the problem space in all major features. Two components that have significantly different algorithmic approaches may conceptualize exactly the same joint domain of work, but have views of that domain for which it is very difficult to define a mapping.

Examples:

- Time points vs. time intervals for scheduling.
- Vector graphics versus bitmapped graphics.
- In Computer-Aided Design (CAD), Constructive solid geometry (CSG) representation of the shape a complex mechanical part versus Boundary representation (BRep) of the same shape.
- Continuous state-based decision making vs. discrete event-based decision making.

Approaches to solution:

In some cases, the view models are non-translatable – there is no mapping between them at the level of abstraction needed for the joint action. In most cases, they are intractably translatable – the function required to perform the mapping is more complex than the joint action itself. In those cases, where they are "translatable," the mapping to a common model for interchange is destructive – it loses potentially important semantics of the producers, retaining just enough to enable a joint action that is approximately correct.

For example, there is a workable translation between edge-detection (event) and level-detection (state) for coordinating machines – an intervening translator buffers event notifications and reports the current net state to the level-detection algorithms, and it compares level outputs between time intervals to generate events for the event-detection algorithms. But the comparison interval has to be small enough to produce timely response by the event-detection algorithms in the worst case, and just long enough to ensure that only consistent states are compared. And when two outputs change in the same interval, the intermediary does not have the information needed to order the event notifications correctly, which may cause improper responses by the event-detection algorithms.

By comparison, the vector-to-bitmap translation can only be performed in one direction – the inverse mapping requires image analysis heuristics – and even in that direction the semantics of the (vector) producer is reduced to a crude representation that may be just adequate to the (bitmap) consumer. And the CSG-to-BRep transforms are intractable – they require significant computation and produce synthetic representations in the other view that may not have the features the designer intended.

### 5.2.2 Conceptual scope conflicts

A *conceptual scope conflict* occurs when a concept that is important to the interaction as viewed by one component is not present in the interface model of the other. That is, one external model for the interaction is a subset of the other, and a sometimes important concept is not in the subset.

Examples:

- A 3-button mouse needs to operate using a protocol originally designed for a 2-button mouse.
- One component manages versions of documents while the other does not have a "version" concept in its document identification model.
- One component requires an approval for the work item associated with the joint action, while the other does not have an "approval" concept.

Approaches to solution:

If the scope of a component is broader than is required for the integrated system, the unnecessary information (and behavior) it produces can be ignored and the unimportant information it requires can be synthesized values that will cause, or not adversely affect, the intended behavior.

(This is what happens with 2½-button serial mice – middle button events are encoded as movements that are zero in both axes.)

If the scope of a component is narrower than is required for the integrated system, the component is only partly integratable into its role, and the missing functionality must be provided by an add-on component or an add-on information set that is somehow accessible to a wrapper.

### 5.2.3 Interpretation conflicts

An *interpretation* conflict occurs when the listener's understanding of the speaker's utterance has different consequences for the listener's subsequent behavior from those expected by the speaker, or by the systems engineer. That is, the technical communication is completely successful, but the intent is not (quite) fulfilled – the message has a (somewhat) different meaning to the listener than it did to the speaker.

In the worst case, the listener's interpretation is completely different from what was intended, and this is usually accompanied by some kind of conceptualization conflict as well. But more commonly, the listener interprets the communication more narrowly than was wanted for the joint action, more broadly, or incorrectly in some important detail, with unintended consequences for the joint action or subsequent actions.

There is no requirement for the communicating resources to have a completely consistent understanding of a term or sentence that appears in the communication. Since each actor has its own concerns, its internal interpretation of a sentence will involve the relationship between the terms used therein and the actor's internal concerns. This interpretation may thus involve concepts that have no meaning to the other actor, because the other actor does not have that concern. In fact, it is possible that, for concerns unrelated to the purpose of the communication and their respective roles in the system, the two actors may make *incompatible* interpretations of the communicated sentences.

Examples:

- One component interprets a given message as a command to perform an action while another component interprets it as a notification that the action has been performed.
- One component's diagnostic dump is interpreted by another component as flight data. (Although this did occur during the Ariane 5 disaster, it was not the cause of the failure [52].)
- Components assume different units (e.g., metric vs. English measure) for measurement values that don't specify the unit.
- One CAD system distinguishes a Point from its Locus, and allows two distinct Points to lie on the same Locus and be separately manipulated in geometric operations; the other does not make this distinction in its internal model. Because it makes no difference in its interpretation, the second system generates an exchange file containing a new Point for each occurrence of the point/locus in any geometric construct. But when the first system reads that file, it interprets all those Points as distinct, and does not, for example, recognize a curve-of-intersection, because the Point objects are not the same ones.

Approaches to solution:

The first two examples above are mismatches of component-to-role, masked by the use of a common interface specification and common terminology for two entirely different roles. This kind of conflict can only be resolved by reconsidering the resource-to-role assignments after a proper model of the design roles of the reusable components has been identified.

The other two examples demonstrate more subtle problems in the specification of the interface and in the corresponding external models of the two components. Such a problem is solved by a wrapper that translates information units and referencers (see 5.2.4) from the speaker to those entities that will best preserve the original semantics when they are interpreted by the listener system. In general, this can be done correctly when the interpretation rules of the speaker are known, unless the speaker's model also has a smaller conceptual scope than the listener's (see 5.2.2).

Interpretation conflicts often occur when standard interfaces are used, because the intent of most such standards is to accommodate several slightly different internal models of the problem space and achieve communication of the "least eommon denominator" semantics. When the standard permits more than the lowest common denominator to be carried by the protocol without providing rigorous interpretation rules, variations in interpretation assumptions emerge.

Interpretation conflicts often occur because of interactions with elements of the internal model of the interpreting resource that have no ostensible relationship to the function of the communication. In the CAD system example above, the first system supports an explicit operation called "anti-aliasing" that will produce exactly the interpretation made by the second system. It is the fact that that operation is *optional* in the receiving system and *implicit* in the sending system that causes the receiver to get the interpretation wrong. And the anti-aliasing operation itself may have no relationship to the *purpose* of the transmission of the geometric model.

### 5.2.4 Reference conflicts

A *reference* conflict occurs when the communicating components use different *systems of reference* for identical concepts — they use different properties of the thing, or different naming schemes, to identify it. The concept in question can be a type of object, action, property or relationship, or an instance of any of them.

In simplest cases, the object in question has more than one "identifier" and the components use different ones. In more complex cases, the basic identifier is the same, but the components have different taxonomies for the problem space and the "complete names" of the objects, or the implicit prefixes for the identifiers, are different.

Examples:

- One component uses supplier-id to identify an organization; the other uses the tax-id. (different properties, naming schemes)
- One component identifies the Part by item number on the order form; the other identifies it by stockroom and bin number. (different relationships, extended properties)
- One component uses Cartesian coordinates to identify a point; the other uses polar coordinates. (different common systems of reference)
- One component specifies supplier "Preference" as "primary," "alternate," and "emergency-only"; the other uses "Ranking" with values 1, 2 or 3. (different model-specific systems of reference)
- One component has the concepts Circle and Ellipse; the other has only the concept Ellipse and represents a circle by having the two foci be identical. (properties vs. names)

Spelling is inextricable from the semantics of reference. A *referencer* (a name or a term) appearing at an interface (a "token in the exchange") is by definition *only* a representation — it has no conceptual value. How it is spelled is what it *is*, and what it *means* is the thing it is defined by the interface to *represent*. Referencers for *instances* of the modeled classes of objects are like the names of people — they have no meaning other than that they identify a particular instance; they are *not* words. In general, it is not possible to translate referencers without knowing the exact relationship between the two systems of reference. And in some cases, such as supplier-id to tax-id above, the relationship can only be expressed by a name-to-name correspondence table. Heuristic mappings between systems of reference may sometimes be guided by recognizing the associations between reference tokens and the natural

language words from which they are derived. But these are special cases that require knowledge that such associations exist.

Approaches to solution:

Construct a wrapper or intermediary to translate one reference syntax to the other, using appropriate mapping rules and possibly "equivalence tables" for the translation of identifiers between systems of reference.

Reference conflicts are closely related to syntactic conflicts (see 5.1.2), and the simplest problems can be solved in the same way, usually as part of the syntax transformations. But the interesting reference conflicts require model-specific reasoning, auxiliary databases or mathematical functions in the wrapper/intermediary to produce correct translations of the referencers.

## 5.3 Functional conflicts

Functional conflicts arise when the behavior of one communicating resource differs from the behavior expected by the other resource in ways that interfere with their interaction, with their successful performance of the intended joint action, or with subsequent interactions and functions of either party.

The functional aspect of integration concerns joint action: Each required function of the system results in a process specification that decomposes the required function into required behaviors of the individual resources comprising the system. For each behavior that involves joint action, there is a set of requirements for the performance of that subtask that must be met by the corresponding separate and interacting activities of the communicating parties. That is, the functional aspect of integration relates to the requirements for the effect of the behaviors of the components and the communications between them on the behavior of the system as a whole.

What is required in the functional aspect is that the utterances produced by a speaker resource and the behaviors the listener resource associates with those utterances (including the listener's responses) will contain no unpleasant surprises for the system engineer. The resources themselves need not necessarily be aware of the effect of those behaviors on the functions of the overall system. But for interoperability between two resources communicating directly, it *is* a requirement that the behaviors of the listener produce the results expected by the speaker, and produce no results that are visible and significant to the speaker, but unexpected.

In general, functional conflicts relate to the suitability of the resources for the system roles that have been assigned to them. And in most cases, resolution of a functional conflict requires replacement, augmentation, modification or reconfiguration of the component. As a consequence, the resolution of functional conflicts is nearly impossible to automate.

In some cases, the reconfiguration of a component can be automated. Indeed, it is possible to build components with configurable behaviors that can dynamically adapt their functionality to be compatible with a particular partner or environment. The adaptability of such a component could be as simple as having a handful of different behavior options, each of which is compatible with the expectations of a particular partner (a "pre-integrated" component). Or it can be as sophisticated as having multiple dimensions of functional reconfigurability corresponding to differences in potentially useful roles for the component, and negotiating the set of roles and the requirements for those roles with similar components to accomplish a system task (a "self-organizing" component).

### 5.3.1 Functional model conflicts

A functional model conflict arises when two communicating resources have incompatible factorings of the system activity space. That is, the two resources are designed to support different behavioral decompositions of a higher-level system function. In many cases, the two decompositions of the function into elementary tasks may be more or less compatible, but the two models make different role assignments for one or more of the elementary tasks. That is, there may be a task that each expects the other to do ("nobody's job"), or a task that both expect to do themselves ("overlapping roles").

Examples:

- Incompatible models: A dispatcher expects to obtain information on the availability of manufacturing resources and to use that information in making job assignments in real time. The manufacturing resource database is kept in a Manufacturing Resource Planning (MRP) system that expects to plan the availability of resources on the basis of future job requirements, maintenance requirements, etc. These two systems have incompatible factorizations of the function "schedule jobs," which leads to incompatible models of their roles and makes it nearly impossible for them to communicate about resource availability.
- Overlapping roles: A Computer-Aided Process Planning (CAPP) system decomposes a complex part description into manufacturing features and uses its knowledge basis to determine the optimal sequence of operations for fabricating those features to make the part. It then accesses another internal database to estimate the time required to perform that sequence of operations on a given machine. The manufacturer's machine simulator can provide a much more accurate estimate of the time required, given the same sequence of operations, but the CAPP system is not designed to communicate with any other source to obtain time estimates. Here the activity factorizations are compatible, but the functionality of the CAPP system overlaps that of the machine simulator.
- Nobody's job: An email exploder expects messages to be assigned Message IDs by the mail relay component. However, the targeted mail relay treats messages lacking Message IDs as invalid and ignores them. It is nobody's job to assign the Message IDs, so these components cannot interact to distribute email.

Approaches to solution:

When the higher-level functional model that the component was designed to fit is incompatible with the actual functional model of the system, integration may be impossible — the component is probably ill-suited to its system role. In the best of cases, a human engineer can invent a work-around using some feature of one of the systems that was designed for a different purpose. In the case of the MRP system above, a common practice is to manually reserve long time periods for certain resources, so that the MRP system makes them unavailable for its planning, and then manually release those resources when the dispatcher is activated, so that they are reported to it as "available." An alternative work-around is to gather all the dispatchable jobs into one big job for the MRP system, let the MRP system schedule all the needed resources for that big job, and then convert the dispatcher request for "available" resources to a request for resources assigned to the big job.

When the problem is overlapping roles, it is sometimes possible to allow one of the resources to perform both the desired activities and the undesired activity, discard the results of the undesired activity, and feed the results of the desired activities to functions of other resources to obtain the desired system functionality. In the CAPP example, this appears to be possible. But this is not possible when the undesired activity will alter the desired results in such a way as to interfere with the proper functions of other resources.

When the problem is "nobody's job," it is necessary to provide an intermediary that intercepts the invocations and performs the missing task(s) before or after invoking the functions of the other resource.

In any case, problems of this kind do not seem to admit of any kind of automated solution.

### 5.3.2 Functional scope conflict

A *functional scope conflict* arises when one party's behavioral model for a function contains more activities than the other party's model, and the activities that are not in the intersection of the two models are important to the *interaction* (not necessarily to the system). And in the general case, neither activity set is a subset of the other — the intersection is large, but each party's model contains one or more activities that are not in the other's model. Functional scope conflicts are commonly associated with Interpretation conflicts (see 5.2.3).

When the requester's model is larger than the performer's model, the performing resource executes a subset of the expected behavior, leaving some expected tasks not done. This is not necessarily a problem for the overall system, if the function of the system is not impaired by leaving those tasks undone at the time of the request. That is, the requester's model might be larger than was needed for the system to function properly.

When the requester's model is smaller than the performer's model, the performing resource executes unexpected activities as well as those requested. This is not necessarily a problem for the overall system functionality, if the unexpected activities are expected by the system engineer and are important to, or at least do not interfere with, the intended function of the system.

Functional scope conflicts are local to the interaction between two resources. They may or may not be related to partly integratable characteristics of one or the other of the resources relative to its system role. But even when both resources are fully integratable into their roles, functional scope conflicts can produce interoperability problems. Although the activities of the performer are consistent with the requirements of the overall system, they are not consistent with the expectations of the requester. And the resource interaction breaks down if the requester becomes confused when it detects differences between the resulting system state and its expectations for the system state.

Note — The "overlapping roles" and "nobody's job" cases of functional model conflict arise when the issue is fitting the component into its system role, i.e., when the overlapping or missing tasks are important to the system. Those problems are functional scope conflicts when they interfere with the interaction of the components themselves.

Examples:

- A relational database system interprets a DELETE operation to delete only the row that represents the object named, but an object-oriented database system interprets a DELETE operation to delete the object named and all the objects that are dependent on it. If the requester was expecting the object-oriented behavior, and the performer is a relational database, objects which should have been deleted will still appear in the database. If the requester was expecting the relational behavior, it may subsequently attempt to make new associations for objects which have been deleted.

Approaches to solution:

When the requester's model is larger than the performer's model, it may be possible to construct an intermediary to intercept the request and expand it to multiple requests to the performer, or to multiple performers, which together accomplish the function intended by the requester. This is a characteristic behavior of distributed database systems, whose function is to automate the solution to a class of problems of this kind.

When the requester's model is smaller than the performer's model, however, very complex work-arounds are usually needed, and the problem may not admit of solution. When the performer's behavior is consistent with the requirements of the system, it is necessary to mask the correct-but-unexpected changes in the system state from the requester.

### 5.3.3 Embedding conflicts

*Embedding conflicts* arise when the behavior of the component is affected by the integrated system in such a way as to produce unexpected and undesirable results.

Embedding conflicts frequently arise during the integration of reusable components with intentionally ambiguous specifications of their functions. The component is intended to exhibit some adaptability in behavior, so that a specified function has multiple detailed interpretations, any of which the component might be configured or conditioned to perform.

In the case of *configuration*, the detailed performance of the functions is controlled by externally accessible configuration parameters that are usually set on "building" the executable form of the component, or on instantiation of the executable process. In this case, the configuration may evince a kind of "emergent behavior" from the component, resulting from the interactions of the configuration parameters with actual events in the system, that neither the designer nor the systems engineer could completely predict. The component must be tuned to produce the intended behaviors for the role.

In the case of *conditioning*, the system tries to learn the usage intent of each client and adapt its behavior to that intent (by adjusting internal configuration parameters) when interacting with that client. There are well-known examples of systems that, based on a sequence of characteristic calls, intuit a larger purpose to the client interactions

that may or may not be correct. "Learning algorithms" based on pattern recognition can produce distorted internal configurations of the component if the initial set of interaction instances is not representative of the scope of the business processes in which the component is used. The heuristic algorithms that use these statistical configurations to make decisions about behavioral details can then produce inaccurate and incorrect results and unexpected behaviors. They can also produce incorrect predictions for subsequent interactions that cause confusion in the joint action when they prove to be wrong.

Embedding conflicts can also arise when the behavior of the component is sensitive to policy aspects, such as speed and accuracy, of the behaviors of other components in the system. If the component does not have sufficiently accurate information at the right time, it may make an incorrect decision. And embedding conflicts can also arise from non-deterministic sequencing of system behaviors. For example, a component might erroneously deduce the occurrence of an event from a difference between two values in a database, when those two values are set separately by other components that are not synchronized with it.

### 5.3.4 Intention conflicts

*Intention conflicts* arise when the component is being used in a way the designer did not anticipate, and unexpected counterproductive behaviors of the component occur. Although the intentions of the component designer may be stated, the corresponding assumptions about, and limitations on, the scope of its behaviors, may not be, with the result that the functional specifications for the component are insufficiently detailed or accurate with respect to the needs of the system engineer for the integrated system. That is, the behavior of the component is not, in some important detail, what the systems engineer inferred from the specification.

Intentional conflicts relate to the clarity and accuracy of functional specifications, but not (necessarily) to the quality of the specifications. Intentional conflicts relate to differences in the details of the specification as understood by the component designer for the intended uses of the component versus the specification as needed by the system engineer for the role of that component in the larger system. That is, the designer of a reusable component cannot know all of the information about the component's behavior that every subsequent systems engineer will need. And the only true representation of all the behavior of the component is the implementation of the component itself.

The specification is necessarily an abstraction of the detailed behaviors of the component. But it is difficult to be sure that the abstraction is at exactly the level needed for the target role. The intention conflict arises when the abstraction masks a partly integratable implementation model, by implying required actions that are not performed, by omitting interfering actions that are performed, or by inadequately specifying qualitative aspects of behavior that affect the success of the integration.

Note — Because any model of a component behavior is necessarily incomplete, problems of this kind are likely to be *created* by automating integration processes.

Example: A PDM system loses some information some of the time when exchanging information with suppliers. The integration engineer used the "Note" feature for all text extracted from some standard field of an exchange file, but the PDM system requires the value of a "Note" to be no more than 1000 characters long. The PDM designer expected the "Note" feature to be used for "annotations" to CAD models, and 1000 characters is a common limitation on "annotations" in CAD systems. The PDM system provides "Document" objects to contain detailed information attachments. And this intent was in the User Guide, but the limitation on the length of a Note was an implementation decision that was not documented.

## 5.4 Policy concerns

Policy concerns relate to aspects of the behavior of the system, its components and its data that are not directly related to the functions that the system or its components perform. Rather they are related to *how well* the system, or a component, performs its functions, and whether that is adequate to support the business processes.

In general, only some policy concerns can be predicted in the overall design of the system and stated in its specifications. Those that are will appear in the requirements for the system. Others will be discovered as integration conflicts or simply observed over time in the behaviors, particularly failures, of the integrated system.

It is not possible to automate the systems engineering activities related to satisfying policy concerns – careful human analysis and judgement is always required. It is possible to provide tooling that supports those activities to some extent. In addition, there are a number of specialized components and toolkits that can be used in implementing solutions to policy concerns. In some cases it is convenient to design solutions to policy issues around the use of those components and toolkits, and in those cases some part of the engineering can be automated.

### 5.4.1 Security

The principal *security* concerns are:

- Does the system protect itself from destructive inputs and behaviors, both accidental and intentional?
- Does the system protect its information from unauthorized disclosure?

Components should:

- check information for errors and inconsistencies that will have serious consequences for the continued operation of the system;
- check information for errors and inconsistencies that will have serious consequences for the business processes the system is supporting;
- check authorization for operations that may have serious consequences for the business processes;
- check authorization for operations that may have serious impact on the ability of the system to support the business processes.

In general, not every possible destructive action can be checked and not every human-initiated or externally initiated operation needs to be. But serious loss or disclosure of business information can often be avoided by routine checking of critical inputs and operations.

In automating integration processes, it may be possible to automate the identification of critical inputs, and/or to automate testing processes that make some statistical estimation of the quality of system security.

### 5.4.2 Correctness, credibility and optimality

This section describes concerns about the quality of data in the system and whether it is adequate to support the business processes.

The *correctness* concern is: How close to the true state of the business and its environment is the information obtained and produced by the system and its components? And how important is the difference between that information and the true state to the behavior of the system and/or the supported business processes?

For single information units, only numeric values can be "close" and have a "margin of error" that can be formalized and sometimes computed. Where appropriate, such margin-of-error estimates can be made and, if necessary, carried with the data. Other simple values are either correct or incorrect — they specify either the true state or some other state.

Complex information sets, however, can be partly or mostly correct, without being correct. And a complex information set can be sufficiently accurate for one decision function but insufficiently accurate for another, because different combinations of information elements have different impacts on those decisions. In general, such information sets do not admit of good measures of correctness.

But the critical question for a given information set is: What degree of correctness is needed to support the business processes? For various practical reasons, many information sets will never be 100% correct, but some degree of accuracy will be entirely adequate for the business decisions that information supports. This requires analysis of the

sensitivity of the business processes themselves to the correctness of information processed or produced by the system.

A related question is: What degree of correctness in each information set is needed for the proper functioning of the system itself (and thereby for the support of the business processes)? This requires an analysis of the sensitivity of the system behaviors to the accuracy of information processed by the system, whether produced by its components or externally obtained.

In both of these analyses, there may emerge measures of "goodness" or "usefulness" of the information, as distinct from correctness. That is, we can distinguish among adequate information, better information, and the best information, in terms of measurable impact on the business process and its results (or on the system behaviors and the quality of business process support). This concept is *optimality*. It doesn't measure the relationship of the information to the true state; it measures the relationship to the effectiveness of the business process. (In some cases, for example, consistency of the information may be more important than absolute correctness.)

In the particular case of information that is externally obtained, or produced by components provided by other organizations, the concept *credibility* is sometimes used as a crude measure of correctness. Credibility is a measure of the reliability of the source, rather than a measure of the correctness of the information per se. Credibility is an estimate of the probability that *any* of the information from that source is correct. Credibility can be based on analysis of past performance of the source, on statistical estimates of the quality and accuracy of information available to the source, or on optimistic or pessimistic estimates of the impact of information from that source on the business processes. In this last case, credibility is related to trust (see 5.5.1).

For some business process elements, correctness of certain information units is essential, and the accuracy of the information produced by a single component is not consistently high enough to ensure success of that process element. In those cases, it may be necessary to incorporate redundant or competing components in the system, to provide multiple independently calculated versions of those information units, possibly coupled with estimates of accuracy or credibility. An additional, human or automated, system component then applies some judgment algorithm to the multiple results to obtain a result with the highest probability of being sufficiently accurate. And this becomes part of the definition of the system and the roles of its components.

### 5.4.3 Timeliness

The *timeliness* concern is: Does the system perform its functions and provide accurate information when that support and information is needed to make the business process run effectively?

The system must execute its functions fast enough to maintain productivity of the human operators and fast enough to maintain productivity of the automata it controls. The system must also make decisions, or provide information to humans to make decisions, at the times those decisions are needed to get the best results from the business process.

Timeliness is related to performance (see 5.5.3), but it is concerned with having operations begin and end at the right times to support the business processes, and with having information that is appropriately up-to-date at the times the business process decisions must be made. Efficient performance does support productivity of humans and equipment. But efficient performance alone does not make timeliness happen — starting tasks with adequate lead time for the available performance does. What performance does is to shorten the lead time and thereby improve the possibility of getting the most up-to-date information for the functions. Arranging to have that up-to-date information delivered is the other half of the timeliness concern.

Timeliness and performance, however, have an associated cost, and timeliness/performance that goes beyond the foreseeable needs of the business process usually incurs an increased cost with no observable benefit. For the integrated system, timeliness and the tradeoffs needed to achieve timeliness are critical factors in the system design. In some situations, timeliness can be quantified, but in other situations, timeliness can only be observed.

### 5.4.4 Reliability

The *reliability* concern is: What are the business requirements for the system to be up and running, and can the integrated system meet those requirements? The requirements for the system to be up and running relate to the requirements for support of the business process, typically in terms of availability, timeliness, continuity, and maintenance of state.

*Timeliness* is discussed above (5.4.3).

*Availability* is related to timeliness. It deals with the ability of the system to meet the demands for the services it provides, at times convenient to the needs of the business. Loss of availability has costs in lost time for other resources and timeliness of results. For example, when a system failure results in a temporary stop on an assembly line, the cost is lost production output, measured in products per hour.

There are two major approaches to addressing availability concerns: *scheduling* the use of the system against maintenance requirements and statistical rates of failure, and *backup* of the system components in some form. In some cases, adequate backup for the required business transactions will require multiple components that share the load and can redistribute it when one of them fails. In other cases, what is required is that copies of critical information contained in the component are also stored elsewhere and alternate paths for its exchange are available when the component or the primary communications path fails. In some cases, all that is required is that there be a means for the human agents to conduct the transactions when some components of the system are not available, and a means for synchronizing the system state with the "offline state" of the business processes when those components again become available.

*Continuity* relates to the ability of the system to finish a task that it has started before it fails, or the ability to resume a task after a failure. The business question in the continuity area is: What is the cost of restarting the task? In the case of manufacturing control systems, a system failure after a task has started often results in destruction of a product lot, and the cost of restart is the cost of acquiring new material and rerunning the lot from the beginning, which may mean many hours of labor and machine time.

In the particular case of integration, there are continuity concerns with respect to the joint actions. Is it important to the interaction of the components that there is continuity in a single "component execution"? That is, if the component fails while a joint action is in progress, and has to be re-started, can the system recover? Can the resurrected component recover the state of the joint action on restart? Or will it have to start over, and then can the communicating partner(s) recognize the restart and recover? Can all the communicating partners negotiate a common recovery state if one of them fails during the joint action? In most cases, such concerns must be addressed in the design of the components themselves; otherwise, it may well be impossible to address them during the integration process.

*Maintenance of state* relates to the ability of the system to retain its information when a component fails. In many cases, that is the ability of the restarted system or component to recover the information that was contained in it at the time of failure. It is not usually necessary for the system/component to recover all of the information it had — some transactions can be rerun or recovered and resumed. What is important is that the state of the system on recovery of a component is consistent and is expressible. That is, when a component fails and is restarted, it is important that the system reverts to a meaningful state, which implies a consistent state of all the components and their information. And it is important that all of the components, especially the human agents, know what that state is, to the extent that it affects the proper continuation of the business processes.

Maintenance of state can be a critical problem in the integrated system, because the mechanisms used to enable the interactions may not enable the components themselves to recognize failures in their communicating partners. The common result is that the state of the restarted component is internally consistent, and the state of the communicating partner is internally consistent, but the system state that is the externally important composite of those states is no longer consistent. Additional integrating mechanisms may be needed to synchronize the component states after a component restart, in order to yield a consistent system state. In some cases, it suffices to recognize that the system state after certain failures may not be consistent, and to force the system to restart and re-synchronize the component states.

### 5.4.5 Version conflicts

Integrated system components are often acquired from a variety of sources, including generic information technology vendors, enterprise/domain application vendors, integrator houses, or other departments in-house. These components themselves may include or depend on other subsystems from sources just as diverse. All of these components are subject to revision, and many with their own independent life cycle. Some of these revisions can cause incompatibilities with their usage. We call these *version conflicts* (some issues leading to these conflicts are discussed in 5.5.5). While these conflicts are often an issue at deployment time as well as over the life of an integrated system, they can also become an issue for an integration project itself if the project spans multiple organizations or requires significant development time relative to component life cycles.

## 5.5 Logistical concerns

Logistical concerns relate to aspects of the behavior of the system and its components that may influence design, but are not directly related to the functions the system performs or even to how well the system performs them. Rather they relate to the non-functional impact of the system on the business processes, and on the business itself.

In general, logistical concerns appear in the requirements for the design of the system but do not appear directly in its specifications. They are included here for completeness. Logistical concerns are commonly addressed using "management" approaches. As a consequence, the idea of automating the resolution of logistical concerns is out of the scope of this paper. Management decision-making is its own discipline and has its own tools.

### 5.5.1 Trust

Trust concerns center on the behaviors of components of the system that are under the control of other organizations. This occurs primarily in ad hoc systems built of autonomous components that attempt to do a mutually beneficial task jointly, such as service brokers, information brokers, and agents for customers and suppliers. Trust may also relate to subsystems that are acquired from software/service vendors and installed in the integrated system and retain some communications relationship with their source, such as remote maintenance.

Trust concerns typically lie in the following areas:

- *Web of trust*: when the system must deal with components that are not under its control, it may rely on a trust authority to authenticate, and possibly characterize, those components. The alternative is to supply all the resources and processes needed to ensure the identity and other characteristics of those components. In essence, the system designer assigns his trust concerns to a third party, which must itself be trusted, usually by both parties or their third-party agents.

- *Credibility*: when the agent provides information to your system, to what extent do you believe that the information is correct? The information could be less reliable because its sources are not very good, or not currently up-to-date, or because the agent's internal algorithms are not very strong — these concerns apply to components in general and are discussed in 5.4.2. But the information could also be unreliable because there is some motivation for the source to provide inaccurate information, or because the business practices of the other organization tend to pollute the information.

- *Disclosure*: when other components of your system provide information to the agent, will that agent protect it in the same way your system would? Are the security policies (see 5.4.1) of the other organization consistent with your needs for your data? The agent may, by accident, design, or simple lack of concern, disclose sensitive business information to unauthorized parties.

- *Abuse*: when other components of your system provide information to the agent, is it possible that the information may be used for a different purpose? In addition to that which is required for the system, the agent may do something that is irrelevant to the system, but serves a purpose of its provider. That something may be harmless, or it may be counterproductive to your business goals.

### 5.5.2 Competition

Competition is a characteristic behavior of some systems in performing one or more of their functions. Specifically it occurs when the design of the system includes more than one component that can perform a given function and allows the choice of component to be made dynamically according to some algorithm based on dynamic characteristics of the behavior of the components. The components themselves may be designed to compete, or may compete unwittingly, but at least one component must be aware of the competition and perform the selection algorithm.

Examples:
- auction systems with bidding agents, where the bidding agents are designed to compete
- job dispatchers and workstations, where the workstations compete unwittingly for job assignments

Competition is designed into a system in order to achieve maximal productivity at minimum cost for some functions of the system and their corresponding business processes. But competition also produces, or can produce, a number of side-effects, some of which may have undesirable impacts. When competition is designed into a system, other elements of the system may need protection from these side-effects.

Examples:
- An agent may send requests to 10 different agents expecting to use the first result returned and ignore all the others. This behavior is beneficial to the process the agent is performing, but it reduces the productivity of the agents whose result is unused and consumes 10 times the processing and communications resources needed for the function.
- A knowingly competing agent may learn the selection criteria and lower the quality of its behavior in other ways in order to optimize the value of the selection factors ("If I decrease accuracy, can I compute Fourier transforms faster than another agent?"). This behavior may support some goals of the provider of the agent, but it might reduce the value of the system in supporting its business processes.

### 5.5.3 Cost

Cost is always a concern in integration projects, as in any other engineering project. Indeed, for many decisions, it is *the* primary concern that drives all the others.

*Cost of integration*: This is the cost of engineering the integrated system: design, acquisition of components, implementation, testing, operator training, etc. This is not a simple linear measure but a multidimensional graph of tradeoffs, primarily of degrees of functionality against return and against the several elements of cost. For example, the more complete the information and functional mapping between components, the greater the business functionality. But the rule of diminishing returns applies to this mapping: at some point the increase in cost cannot be justified by the marginal improvement in the business return.

*Cost of operation*: The cost of operation is the cost of running the integrated system. This is usually phrased in terms of the business resources it consumes. With respect to the engineering problem, we used the term "resource" to refer to the software, hardware and human components that were engaged in providing the functions of the system itself. The business resources consumed by operating the system are the hours of use of the personnel to operate and maintain the system, and the hours of use of the hardware systems that are attached to the components and consumed by operation of the system, whether or not they are actually involved in performing the functions of the system. This includes the computer systems and the specialized display, measurement and manufacturing systems which are inaccessible or limited in their use for other tasks when the system is operating. For example, when a machine controller is busy simulating a possible future operation, and the machine itself is idle, both resources are consumed, even though only the machine controller is being used for the system functions. Similarly, when a computer system connected to a vector graphics display is performing an integrated system activity and a computer-aided-design activity at the same time, with the consequence that the response time for the design activity is noticeably increased, the cost of operation of the integrated system includes the lost time of the design engineer and the vector graphics resource.

Note — The *productivity* of a system is the ratio of its business accomplishments over some time period to the total resources it consumes, directly or indirectly, in that time period, or equivalently to its cost of operation.

*Cost of change*: Business processes tend to undergo minor changes fairly frequently, in response to changes in business opportunity, market, technology and law. As a consequence, the integrated system will need to respond to such minor changes, in order to maintain its usefulness in supporting the business processes. The cost of change is the expected cost (based on prior experience) of the analysis, re-design, re-programming, and testing of the system, and of the re-education of users, needed to support the minor changes in the business practices. A seemingly cheap and successful integration can turn into a expensive failure if the original integration selects a mechanism that cannot easily be modified to accommodate minor changes. For example, the integration project develops software for simple data-element-to-data-element mappings rather than more generic model-based mappings that could be extended to additional fields or different data formats; and 3 months after the system goes into operation, the new version of one of the components has reorganized and extended some of its output data. Cost of change is closely related to flexibility (see 5.5.4).

*Opportunity cost*: The choice of business practices and the design of the systems to support them will create the capability to take advantage of some business opportunities, and simultaneously remove the capability to take advantage of others. The potential business value of opportunities lost by the selection of capabilities in the system design is called the *opportunity cost*. Dealing with opportunity cost may create significant tradeoffs in the power and complexity of the system versus its expected value for the immediate business concerns. It is very expensive to build a system which can support a dozen diverse business practices. Sometimes this is necessary, because being able to support only certain practices will have significant impact on the ability of the business to compete in its target markets. But when the expected loss of opportunity is low, design efforts to avoid that loss should also be minimized.

### 5.5.4 Flexibility

Flexibility refers to the ability of the system as-integrated to support minor changes to the business process outright (e.g., by changes in data elements and usage practices) or to undergo minor modifications to be able to support the modified business process. Some flexibility can be designed-in in the integrated system, based on prediction of additional requirements for capabilities that may arise in the future.

The test of flexibility is the time and cost required to make the changes needed to address foreseeable requirements for change in the business processes. Conversely, there may be a significant cost for achieving the flexibility to meet those requirements in terms of time and cost of building and deploying the system to meet the immediate needs. But in many cases, adequate flexibility to meet the most probable needs can be built in at low cost, if an effort is made to identify and estimate the likelihood of these future requirements.

### 5.5.5 Autonomous change

When the integrated system involves components provided by other parties, which are occasionally upgraded to accommodate other uses, the system must be able to accept, or undergo minor modifications to accept, minor changes in the behaviors of the components.

One aspect of a system that affects its ability to respond to these sorts of changes is the *openness* of its subsystems. When an independently supplied component is integrated into the system, we may treat it as a "black box," and rely entirely on its published specifications, or even its support of standard interfaces, to obtain the desired behaviors. We refer to such a system as *open* because its intended function and other integration related behavioral aspects are documented and made openly available. Integrators and other component designers may then use this documentation to build compatible systems. In other cases, we may treat an independently supplied component as a "gray box," using other evidence for its expected behaviors, such as previously observed behavior, reported behavior, intuited behavior, or reading of code (or even reading of bug reports). In many cases, for successful integration the engineer must make these inferences regarding the behavior of the system components.

We refer to these gray box systems as *under-specified*. However, the optimal level of specificity may be elusive. As we have described earlier in this section, even complete interface specifications do not always provide all the information needed to eliminate integration conflicts. On the other hand, over-specified component behavior may lead to exploitations of those details and unnecessary dependencies on them, making the system very sensitive to component changes ("brittle").

The level of specificity of a component tends to lead to tradeoffs with respect to a system's ability to respond to different sorts of change. Incorporation of a gray-box component into a system (thus depending on its undocumented behavior) typically imposes a higher risk of increased cost due to unexpected change — in the costs of checking for it, discovering it, and reacting to it (and possibly cleaning up any damage to the business processes caused by it). On the other hand, an open component implies a certain agreement between the component designer and the component's users. This limits the freedom of the component designer in responding to changing needs, but gives the system integrator a certain level of confidence that the integrated system won't be subject to major perturbations from changes in that component, as component versions, and sometimes even component suppliers, change. However, even published interfaces and specified functionalities may change. So it is still necessary to check for and deal with changes in a new version of a system component, in order to avoid unexpected impact on the business activities. Depending only on the documented interfaces and behaviors of open components only reduces the cost of autonomous changes in components; it does not eliminate the cost, or the concern.

# 6 Useful methods and technologies

In this section, we present an overview of several engineering methods and information technologies that may be of value in developing automated solutions to some integration problems.

## 6.1 Systems engineering

Systems engineering is discussed here because it provides a body of knowledge directly concerned with the issues of engineering and integrating complex systems. Though the discipline of systems engineering has its roots in hardware-intensive systems, the underlying principles apply as well to software-intensive systems.

Systems engineering is any methodical approach to the synthesis of an entire system that (1) defines views of that system that help elaborate requirements, and (2) manages the relationship of requirements to performance measures, constraints, components, and discipline-specific system views.

The key terms in this definition of systems engineering are *requirements* and *views*. Requirements are optative statements that are intended to characterize a system. Views are conceptual images of the system that embody the concerns of a particular viewpoint (see Section 4) and omit others.

Systems engineering is foremost about stating the problem, whereas other engineering disciplines are about solving it.

What systems engineering can provide towards automating software systems integration is a plan for organizing information and resources and directing activity that should accomplish the integration task. This plan is called a systems engineering process.

A systems engineering process should:
- identify relevant viewpoints, capabilities, and problem solving resources;
- analyze requirements so as to identify a correspondence of these to system functions, problem solving agencies, and performance measures;
- charge problem solving agencies with the task of meeting particular requirements;
- evaluate performance measures and constraints, so that success can be recognized.

A systems engineering process is implemented when software systems integration is automated. In this paper we described, in general terms, the activities and information flows of systems integration (see Section 3). What

remains (and is beyond the scope of this paper) is to express those activities and information flows in operational terms. Specifically concerning the viewpoints and impediments to integrations discussed in Sections 4 and 5, the bulleted list just provided can be refined for that context.

A systems engineering process for software systems integration should:

- identify viewpoints relevant to the integration aspects defined in Section 4, and identify existing models or tools and techniques for developing and representing the needed viewpoint models;
- identify automated tools that are capable of resolving the concerns and conflicts described in Section 5 that are relevant to the integration problem at hand;
- analyze role requirements and interaction requirements and render them in a representation that may be shared among problem solving agencies, and transformed into partial specifications;
- identify tasks and orchestrate the execution of tools;
- develop a validation suite that exercises the solution and measures the results.

In the context of software integration, we distinguish the *system concept* (an abstraction) from a *system solution* (a representative instance of the system concept). For automating integration, the challenge with respect to requirements is to state them formally, in such a way that they characterize the system concept, do not presuppose a system solution, and yet still enable automated means to identify solutions.

The available and emerging systems engineering techniques include:

- standard practices in formulation of requirements
- standards for representation of requirements
- common practices in evaluating and prioritizing requirements
- common practices in aggregating and refining requirements
- common practices in segregating requirements and allocating them to viewpoints
- tools for capturing requirements, derived aggregates and refinements and the relationships among them, and evaluations, priorities and assignments to views
- tools and practices for analyzing requirements interaction and interference

The tools support particular analytical techniques and knowledge bases, many of which are viewpoint-specific.

It should be noted that systems engineering is an immature engineering discipline in many of the same ways as computer science. Though the management of requirements is essential to systems engineering, a best-practice in requirements engineering has not yet been established. Further, classifications of requirements vary (necessarily) from one project to another.

## 6.2 Design engineering

Fundamentally, systems integration is a design problem, and automated integration is a design automation problem. Design engineering is a disciplined practice that results in the specification of a system that satisfies stated requirements. Integration is a design activity that identifies how interactions of subsystems enabled by particular communications can satisfy functional requirements. Integration is design with the restriction that the range of possible behaviors from the component subsystems is fixed.

The design engineering process is a recursive sequence of four basic activities: decomposition, synthesis, composition, and analysis.

A decomposition phase takes a functional or behavioral requirement and converts it into a set of sub-behaviors that together will accomplish the required function/behavior. Each sub-behavior is assigned to a particular "device." The objective of this activity is to identify sub-behaviors that can be accomplished by a known (class of ) device, or sub-behaviors for which the device can be synthesized. In some cases, some of the sub-behaviors will be further decomposed, giving rise to multiple levels of decomposition. In each case, the requirements that relate to the function/behavior beget refined requirements on the sub-behavior. Additional requirements may be imposed on the device out of its need to work with other devices, and within other parameters of the system, to accomplish the target function.

A synthesis phase identifies, or constructs, a set of device designs that produce a desired behavior. In the lowest levels of decomposition, the device is an elementary component, and the design is either a selection from a catalog or the specification of a shape and material to meet the requirements. (In the case of software, the lowest-level "device design" is the specification of the routine that performs the desired function.) In all other levels, the synthesis process designs the interoperation of the component devices to produce the target behavior from their sub-behaviors.

A composition phase identifies opportunities to "integrate" multiple sub-behaviors in a single device, often by combining elements of two or more component device designs. This produces a single device design that implements more than one sub-behavior. In many cases, these sub-behaviors will be parts of different higher-level behaviors or functions.

An analysis phase tests device designs against requirements and also evaluates them against "non-functional" concerns similar to those identified in 5.4 and 5.5. It eliminates alternative designs that do not meet requirements and those that are unacceptable with respect to other evaluation concerns. It also addresses tradeoffs. The result is not necessarily a single "best" design. It may be a set of "admissible designs" with different evaluations. In synthesizing the higher-level device of which this device is a component, more than one of the admissible designs for this device may be considered. In some cases, the set of admissible designs will be empty, which indicates a need to refine/correct the requirements for one of the sub-behaviors (and the corresponding component device), or a need to reconsider the decomposition of the behavior being analyzed.

The ordering of these phases, the choice of breadth-first vs. depth-first search of the design spaces, and other design process decisions are dependent on the particular product domain, behaviors, and the expertise of the engineer. And engineering organizations may enforce formal breakdowns of a large engineering task, assigning different components and different phases for the larger subsystems to different engineers. So one can argue, for example, that "reuse of components" is part of synthesis, or part of composition, or just a consideration in analysis, depending on how these practices are distributed over the engineers in a given organization.

It should be apparent that there are many parallels between this process and the systems integration process described in Section 3. The decomposition of function to sub-behaviors (process with activities) with component devices (resources) implementing those behaviors (playing roles) is common. The synthesis phase is identifying resources and fitting them to roles. The analysis phase is identifying the interaction constraints, the conflicts and the tradeoffs. While software engineering doctrine generally dictates against the composition phase in designing software components, it is common in systems integration to find a single large subsystem that implements several sub-functions/behaviors of different higher-level functions and processes.

The primary lessons from the design engineering process itself are:

- Multiple alternative designs should be considered, and tradeoffs evaluated; the first viable solution isn't necessarily the best.

- Designs must be tested against the requirements (and other considerations), and these tests can take place at several levels of decomposition if the requirements are properly refined.

Automated systems that might choose between competing designs lack the intuition of their human counterparts and hence cannot discard low-potential alternatives out-of-hand. An automated system that has itself synthesized the design may have a basis for evaluating the design as part of the knowledge base that directed the synthesis process. But when the automated system was not responsible for design synthesis, it can only determine acceptability of the design by performing well-defined tests. In either case, the automated system must simulate by deterministic algorithm the non-deterministic decision process performed by the human designer. For example, the human design process may be simulated with "optimistic" algorithms that search the design space by generating and testing potential designs and backtracking from bad choices [117].

It is the process of generating and testing potential designs that is difficult, and particularly so in software design. Consider the resources available to the design engineer, versus those available to the software system engineer.

The tools of the design engineering process are:

- component catalogs, with common classifications of behavior, common description languages, and common specifications for important and distinguishing properties
- knowledge bases that relate off-the-shelf components and previous designs to functional requirements
- toolkits for performing well-known technical analyses of design properties and behaviors
- formal policies and sometimes formulas for evaluating the non-functional evaluations and tradeoffs

In comparing these to the tools for the software engineering process:

- There are few software equivalents to component catalogs. As to common description languages, such that terms used in one specification have the same meaning when used in a specification for an alternative component, there are some prevalent vocabularies for some applications. And there does not seem to be agreement on what the distinguishing properties of many off-the-shelf systems are. But it is possible that any of these could be developed.
- It is possible to develop knowledge bases that relate certain functional and behavioral requirements, particularly technical requirements for communication, to components and design patterns that meet those requirements.
- The only well-known technical analyses of software design properties and behaviors are tests for conformance to certain standards for technical interfaces, but there are toolkits for those.
- The formal policies and formulas often exist. For automating the engineering process, it is necessary to produce machine-interpretable forms of these.

## 6.3 Specification and modeling

Models constitute a principal element of the design of business processes and software systems. The existing models of processes, systems and components provide the only existing formal representations of the information on which the integration process depends. This section addresses the nature of such models, the information that is captured in them, and the utility of those models in automating integration tasks.

A model is defined in a *modeling language*, which may be textual, graphical, or mathematical. A model has concepts, well-formedness rules, and interpretation rules that are derived from the modeling language and approach. Models are composed of *model elements* defined in the language. Instances of model elements make assertions about the things they represent. These all represent formal captures of some aspects of the semantics of the problem space.

All models have viewpoints (see 4.1). One of the issues in integration is to characterize the relationships that occur among the models that contribute to the specification of the system. Doing this requires an environment in which models in the various modeling technologies can be asserted and interrelated. Characterizing the distinctions possible in a modeling language, and characterizing what is being expressed in that language, are key to the success of automated methods that match behavior requirements to capabilities, relate information needs to information availability, and define the coordination requirements for a given interaction. Zave and Jackson [69] describe experiences with a similar approach. They use the terms "viewpoint communication" and "viewpoint composition" to describe it. They suggest that this "communication" makes more explicit the traceability to requirements and identifies conflicts that might exist between models. They also suggest that a set of primitive modeling predicates alone are not sufficient to gain the benefits, and additional predicates are needed to solve any given communication/composition problem. These additional predicates might be (or might be based on) a custom designed, very small meta-model for the particular sort of business being modeled.

### 6.3.1 Functional modeling

Modeling a function or process covers such system aspects as functions, activities, information flow, events, interactions, time, and resources. There are several different approaches.

*Functional modeling* identifies all the functions a system is to perform, the inputs they require and the outputs they generate. It may also include decomposition of a primary function into a *composition* of sub-functions, with the additional purpose of identifying the *elementary functions* of the system. This technique is usually used in defining specifications for algorithms to accomplish the *purpose* of a process.

*Activity modeling* represents a process as a partially-ordered graph (in the simplest cases, a sequence) of interrelated tasks, each having inputs and outputs. Each major activity decomposes into its component interrelated subtasks, and so on, until tasks that can be directly implemented (or directly supported) by software have been identified. A function of the system is a *scenario* describing the graph of actions and information flows resulting in the performance of that function. Relationships among subtasks can be modeled according to time dependencies, or according to input/output dependencies, or both. This technique may be used in modeling an existing process that is currently executed primarily by human labor or by independent systems. An activity model models what the components *do*, not what they are trying to accomplish. This is, however, a solid foundation for evolution of independent systems into an integrated complex, on the assumption that the individual systems reliably perform their specified tasks.

*Process modeling* is a specialized form of activity modeling that depicts the response of a system or component to specific inputs or events, identifying the actions it takes in response to the stimulus and the results those actions produce. Process models in this sense are much more specialized than activity models and typically rely on the existence of one of the other models to determine the context for the detailed behavior specifications. Such models can be used to produce very clear engineering specifications for the behavior of interacting systems at interfaces.

The remainder of this section describes common methods for modeling functions, activities and processes.

### *Programming languages*

Programming languages have traditionally been the standard form for representation of functions in the software community. A description of a function or process written in a programming language is the machine that can execute that process and realize that function. The problem, of course, is that programming languages are about *how* to perform a function, and two programs that perform exactly the same function may be entirely different, since there may be many equivalent algorithmic and mathematical formulations of the same function.

Programming language compilers are capable of verifying that the signature of a function — its name, its scope, the data types of its parameters, and the exceptions that it may throw — is consistent with its invocation. However, they are not yet capable of verifying that the behavior of a function, as defined by its source code, is consistent with its usage.

### *Formal methods in programming*

Many in the computer science community have sought a way of capturing the behavior of functions in a manner that is formal enough to enable reasoning about the behavior of a constructed system, complete enough to express interesting behaviors, and abstract enough to be better than programming language code. The result has been a plethora of formal methods and notations based in mathematics and logic that can be used to specify behaviors. Z [49] and Vienna Definition Method (VDM) [46] are representative examples from the mainstream of the formal methods community. However, description logics, from the artificial intelligence community, have been used to similar effect [42].

To specify the behavior of a function is to specify part of the meaning of the interface as it would need to be understood by any prospective user or integrator. Techniques for formal interface and service specification could therefore be of use in ensuring that the results produced by an automated integration tool were appropriate. In the ideal case, given a formally expressed, desired system behavior, an integration tool could use the specifications of the behaviors of individual components to *prove* the desired system behavior and thus determine a correct integration (see Section 6.5.2, automated theorem proving). A necessary condition for this to work would be for all of the specifications to have been constructed using the same ontology; that is, the interpretations of the terms used in the specifications themselves would need to be consistent. Nevertheless, even if the ideal is not achievable, formal specifications could at least enable a more rigorous characterization of the relationship between software components and the functions that they are believed to perform, using a set of terms that have formal definitions. Further study would be required to determine if the available formal languages are sufficiently expressive for that purpose.

### *Program Evaluation and Review Technique (PERT)*

PERT was initially developed as a means of breaking down large and complex projects into manageable component activities whose resource and time requirements could be estimated with minimum, maximum and expected values, so that the time and resources for the entire project could be estimated with some standard deviation. PERT models characterize subtasks as having required predecessors and successors, but allow many subtasks to exist in parallel, so long as they have no interdependencies. This allows critical paths to be determined and project duration to be estimated accordingly. PERT orders subtasks into a directed graph in which the root node generates the final output of the major task and each arc between nodes is implicitly labeled by an object that is the output of the source and the input of the destination. PERT models assume that an output must be complete before a task to which it is input can be begun. This defines a partial ordering of all subtasks, indicating which ones can be performed in parallel and which must be sequential.

Extended PERT models, such as the Work Breakdown Structure (WBS) and the Critical Path Method (CPM), add information about resources and deliverables to the tasks and allow activities to generate multiple deliverables with different milestones. They allow the start of a dependent activity to be based on a milestone/deliverable that may be distinct from completion of a previous task, and they permit variable commitment of a resource to an activity across time frames. Extended PERT modeling systems may also support formal hierarchical decomposition, where a subtask in one model becomes the major task of another model. All of these modeling approaches are discussed in Modell [31].

These languages are commonly used for business process models, but primarily for one-off projects, such as an integration project, and rarely for the recurring processes the integrated system will support. And their primary thrust is management of time and resources. While PERT(-like) models are a common part of integration projects, therefore, they are not likely to be of direct value in the automation of the integration processes.

### *IDEF0*

Integration Definition for Function Modeling (IDEF0) [12], formerly the Structured Analysis and Design Technique, is actually an *activity* modeling approach that models subtask relationships solely in terms of inputs and outputs, with no concern for timing. It does, however, distinguish three types of input to an activity:

- *input* objects and information, which the subtask processes or consumes to produce its output
- *control* objects and information, which condition the behavior of the subtask in performing the process
- *resource* objects and information — resources used or required by the process, whose availability may affect the timing of the process

IDEF0 supports hierarchical decomposition. IDEF0 makes no assumptions about the completeness or finality of the *input* objects flowing into a subtask, but *controls* are assumed to be fixed for any instance of the activity, and should include the specific stimulus that causes the activity instance to occur. As a consequence, it can be used to model continuous, interacting and cyclic processes, such as the long-term and recurring processes that are the context for systems designs.

IDEF0 models are likely to be a primary form in which business processes are captured, and may serve as part of the formal models of the process to be supported that would be an input to an automated integration tool.

### *Petri nets*

The Petri net [35] is a formal functional modeling method for capturing the flow of control and information in a concurrent asynchronous processing environment. It sees each component as performing a single function with a definable result. The process function is decomposed into a directed, but not necessarily acyclic, graph of component functions in which an arc connects the function that produces a given partial result to each function that operates on that result. Petri nets thus model the functional dependencies inherent in the component functions of a process, and as a consequence, the timing dependencies inherent in their implementation. At the same time, it allows independent functions to occur in parallel, which permits it to model concurrent asynchronous processing of such functions.

Petri nets are commonly used to define the functional dependencies of a process, and thus to identify flows and critical paths. They are also often used to simulate processes in order to obtain rough time estimates, identify bottlenecks, and analyze responses to perturbations in inputs or resources.

Petri nets provide one of the few formal means for capturing a function as the net effect of its distributed sub-functions. As a consequence, a Petri net model might be one of the inputs to an integration automaton, although its primary value may lie in analyzing the expected results of an automated integration.

### *Finite State Automata*

*Finite state automata* (FSA) are process models, rather than functional models. The class includes a large number of published methodologies, variously called *state tables, transition networks, state charts, ladder logic*, among others. The Harel Statechart [97] and Moore Transition Matrix [98] techniques are the most commonly used. In all of these methodologies, each system or subsystem is modeled as having a set of *states*, a set of rules for the transition from one state to another.and a set of actions to be taken, or results to be produced, during each transition. The rules for transition are based on *events*, i.e., detected changes of state in external and internal objects.

"Extended" finite state automata allow the overall state of the process to be decomposed into multiple variables and multiple sub-machines, so that the true state of the whole system is a combination of the modeled states. Extended FSA also model actions that create events and change values (states) of internal objects and external objects. In addition, they can model *derived* combinations of states and events, which may be used to control transitions.

FSA models are particularly useful for formally describing the response of a system to unpredictable situations and events.that is, situations in which the proper *sequence* of actions cannot be readily described.

FSA models provide one of the best understood means of formally modeling the actual or desired behavior of a component, or the behavior of an interaction. Consequently, FSA models might represent the major form of input to a tool that captures interaction and coordination models and automates the construction of the corresponding application "wrappers."

### *Rule-based models*

*Rule-based models* describe the functions and behavior of systems as a set of *situation-response sentences*, i.e., constructs of the form: WHEN <conditional expression> THEN <result statement>.
The <conditional expression> is some Boolean combination of events and "facts" – tests on information available externally and internally to the system – that defines an instantaneous state in which the rule applies.
The <result statement> is a combination of actions the system must take, and results the system must produce, when the rule applies. In any given instant, more than one rule may apply. Such methods often include both functional and behavioral notions.

There are many available languages for capturing and exchanging such models, none of which is in wide use. The earliest and simplest form of rule-based models was *decision tables*, and the more recent formalizations of *business rules* [90][139], are typically rule-based models, at least in part. Rule-based models differ from finite-state automata in that they lack formal concepts of state and event, and need not be deterministic.

Rule-based models are an alternative to finite-state automata for specifying the behavior of components formally, and they may be useful in specifying the behavior of interactions or protocols. But for integration support tools, rule-based models are most likely to be used in expert systems (see 6.5.1).

### 6.3.2 Service and Interface specifications

*Service models* are weak behavior models that describe a component in terms of the distinct functions it can perform, the information required to perform each function, and the information produced on completion of each function. They assume that a function is performed in response to an explicit request, either from another system or from a human user. The definition of a function in service modeling languages is usually limited to its name and its "signature," that is the names and data types of inputs and outputs.

Most service models use a one-request/one-response paradigm, although there may be several kinds of responses for a given kind of request. Such a system performs only one function at a time, and only responds once to any given request, on completion.

Services are modeled using an *interface definition language*. Several of these are in common use:

Java [110] and C# [126] are programming languages frequently used to develop software for remotely accessible services. It is common in those communities to specify the service interfaces in the programming language itself.

The OMG/ISO Interface Definition Language (IDL) [61] is an abstraction of the interface specification capabilities of several programming languages, and it is characteristic of IDL compilers that they can generate the equivalent formulations in the target implementation languages (Java, C/C++, Ada, etc.). Similar languages are used in the X/Open DCE [107] and Microsoft DCOM [108] environments.

All of these IDLs assume the following rules:

- A particular system – the *server* – offers a specific set of services (functions) to other applications. The behavior of the application that uses the service – the *client* – is implied or unspecified in the language, although it may be specified in surrounding text.
- The set of services is described in a single *interface specification schema*, along with all information and objects necessary to specify the services.
- Each service is modeled as a procedure call, which includes the name of the service, a list of inputs and their data types, a list of the principal output results and their data types, and possible alternative responses, together with their associated results.

The Web Services Description Language (WSDL) [79] presents a somewhat different view of the service interface. It defines *message types* with names and required and optional content specified in terms of named information units and their datatypes. A message can be classified as a notification, a service request, or a response to some request message type or types. Messages can be collected into a *port* (an interface to a component) which attaches to each message an indication as to whether that port can send it or receive it or both. Ports can be paired to define a client-server contract, with explicit request-response rules. More generalized forms of n-party contract with rules for which party sends which messages to whom are possible. So any interaction specified in any of the earlier interface definition languages can be specified in WSDL, but more general interactions can also be specified and more information can be provided.

The important characteristic of these specification languages for integration purposes is that they define the information that can be obtained from, or provided to, the components the specified interface, and the means by which that information can be obtained or provided.

### 6.3.3 Object and information modeling

In describing the functions performed by a system, it is necessary to identify the objects and information on which the system acts, and the objects and information it produces. When the process is decomposed into separate activities implemented by separate components, further details of the objects and information, including new intermediate objects, become a part of the specification. Thus models of the *shared* objects and information of a system, called the *universe of discourse* of the system, become an important part of the engineering specifications, and are especially critical to integration.

The universe of discourse has four elements: objects, relationships, properties, and operations. Various modeling methodologies address some or all of these elements in somewhat different ways.

An *information model*, or *conceptual schema*, is a formal description of the possible states of the objects, properties, and relationships within a system. *Information analysis* is the process by which these objects, properties, and relationships are discovered. A *data model* is a formal description of an organization of information units conforming to an explicit or implicit information model.

*Object-oriented analysis* is the process of identifying the objects and operations in a universe of discourse, and then classifying objects by the set of operations they support. The information units that are attached to an object, and the relationships among objects, are then determined from the need to support these operations. An *object model* is a formal description of the object classes and the operations they support, and usually includes the required information units and relationships.

The remainder of this section describes popular methods for information modeling and object-oriented modeling.

#### *The Entity-Attribute-Relationship method*

The Entity-Attribute-Relationship (EAR) method [91] is the oldest accepted information analysis method. It places each element of a universe of discourse into one of the following categories:

- *Entity*: any interesting object
- *Value*: an information unit having a simple representation
- *Relationship*: an association between two or more entities
- *Attribute*: an association between an entity and a value

Entity *types* are distinguished by the set of attributes and relationships the member entities possess. More advanced EAR models attach particular significance to the relationships "is a part of" and "is a kind of." The latter relationship is also referred to as a *subtype* relationship.

EAR methods were closely associated with the development of relational databases, and as a consequence, often have problems representing multi-valued attributes and relationships (situations in which an entity may have the same type of conceptual association to more than one entity or value), which cannot be represented by a single value in a column in a relational table. This gives rise to *value structures* (sets or lists of values), and to *reified relationships* (entity types which represent associations as objects).

EAR methods lead to a number of model representation languages, both graphical and textual. Two of these have been standardized and are in common use:

- IDEF1-X [13] has both a standard graphical representation and a less frequently used textual representation. The graphical format is most frequently used for the publication of models.

- EXPRESS [20] has both a standard graphical representation (EXPRESS-G) and a standard textual representation (EXPRESS). The latter is far more commonly used, and in fact the graphical form cannot capture all the relationships representable by the language form.

Models in these languages represent a common form in which input and output information sets are formally documented, particularly in the case of engineering specifications for manufacturing products and processes.

### *The Object Role Model*

The Object Role Model (ORM) [99], formerly called the binary[6] relationship method or the Natural-language Information Analysis Method (NIAM) [100][101], is an information analysis technique which classifies each element of a universe of discourse into one of the following categories:

- *Entity type*, or *Non-lexical object*, representing a real or conceptual thing that is described in the model
- *Value type*, or *Lexical object*, representing a thing that is used to describe an entity
- *Fact-type*, representing a potential relationship between an entity type and zero or more other entity types or value types.

When a value describes an entity, that is a *fact*. The admissibility of that fact in the model is expressed as a fact-type between the entity type and the value type, having the form: *subject-verb/role-object*, and usually also: *object-inverse role-subject*. When two entities are related in some way, that is also a fact, expressed in the model as a fact-type linking the two entity types. Like a natural language sentence, a fact-type can have multiple participants in different roles.

The ORM allows the modeler to specify whether a participant can have one or more relationships of a given fact-type, whether a relationship is required for all entities of a type or only permitted, and whether certain relationships can only exist in the presence of others. (Relationships that themselves have properties, however, must be *reified* into entities.) Like EAR methods, the ORM attaches particular significance to *subtype* (is a kind of) relationships.

This form of model is commonly used in specifying information bases for business applications, but the standard representations are graphical, and in many cases, their content is not readily accessible to an automated integration tool.

### *The interpreted predicate logic (IPL) method*

The Interpreted Predicate Logic (IPL) method [111] models the universe of discourse as consisting entirely of objects for which certain *propositions* hold. The conceptual schema consists of a set of *sentences* made up of:

- Terms and variables (entity instances, classes, and names)
- Predicates (verbs, relationships)
- Logical connectives (IMPLIES, AND, OR, etc.)
- Quantifiers (all, at least one, at most one, etc.)

Mathematically, the IPL model is a *first order functional calculus*, and has proved in practice to be capable of modeling all possible valid states of objects and information. Additional (modal) logic concepts are required if the approach is extended to model dynamic behaviors.

Languages embodying such methods have been used to capture information for a number of integration activities. The results, however, are not necessarily very readable. In general, the interpreted predicate logic approach to object modeling is an application of the knowledge representation languages discussed in 6.4.

### *Object-oriented methods*

The term *object-oriented* is applied to many different methods whose objective is to develop some form of object model. True object-oriented analysis methods concentrate on the identification of the conceptual objects in a universe of discourse and classify those objects by the set of basic operations they support. The type of a class is established by the ability of its members to support a certain set of operations.

Operations themselves are considered to have two components:

- the *message*, sometimes called the *operation signature*, which identifies the nature of the operation, the types of its parameters, and the types of its results

---

[6] The term "binary" is historical, in that the earliest proponents sought simplicity by requiring all relationships to have the subject-verb-object form. As the method has developed over the last 15 years, unary and n-ary relationships have become accepted.

- the *method*, which specifies the detailed behavior (the actual implementation) of that operation on a particular object

By definition, a class supports a common set of messages. One class may be described as a *subclass* of another, and it is said to *inherit* all operations from the base class, that is, every object in the subclass supports all operations of the base class, and typically has additional operations as well. But object modeling methodologies disagree on whether a class and its subclasses must also support a common set of methods.

Object modeling methods also disagree as to whether a class can be modeled as a subclass of more than one base class. This concept is referred to as *multiple inheritance*. The underlying issue is whether a model implies that classes not described as having a common inheritance relationship must be disjoint. Such a rule is convenient for implementations, but requires unnatural models of some real-world situations.

The properties and relationships of a class can be deduced from the set of operations the class must support. In many cases, the requirement to have a particular property is made evident by an operation that returns the value of that property. The representations of properties and relationships in an implementation are chosen to optimize the methods that use them. They are said to be *encapsulated* in the object implementation, and are of little interest to the model.

Many object modeling methods distinguish between *primitive types* (the types of machine-representable information units), and *abstract types* (the types for which operations are modeled). This distinction gives rise to rules about what is encapsulated and what is not, and what types can appear where in messages.

The principal object-oriented modeling methodologies are named after the authors of the books in which they were introduced — Rumbaugh [92], Booch [93], Schlaer-Mellor [94], Jacobsen [95]. In the late 1990s, however, a group of the prominent authors and tool vendors came together to create a standard object-modeling language, called the Unified Modeling Language (UML) [80], which is now in common use. The UML "static structure diagram" is a conventional object-oriented modeling language, with a few minor additions. But UML also includes a number of other graphics languages that borrow from each other's concepts, including languages for process and activity models, information flows, *use cases* (scenarios), and a number of implementation-oriented modeling concerns.

UML models, which have a standard semantic foundation and a standard exchange form (XMI [81]) are likely to be the primary form of formal specification of the interfaces to components, and they may also be used in defining information sets and specifying some behaviors.

### 6.3.4 Meta-models and model integration

As modeling languages have become a significant element in software systems design, both software engineers and their tools are expected to use the models as prescriptions for the corresponding software elements. This creates significant pressure to formalize the semantics of the modeling languages, which are captured in the text descriptions in the language manual, in order to ensure that the model is consistently interpreted. A related objective of this effort is to permit a translation, or a partial translation, of the information from one model into another, which may be represented in a different language. This makes it necessary to have a formal model of the semantic concepts of the modeling language. Such a model is said to be a *meta-model* for the modeling language.

We examine meta-models here, because they may provide a formal basis for interpreting a modeling language, and therefore a means for interpreting the models specified in it.

There are two common approaches to the construction of meta-models: use of the modeling language to model its own concepts, and use of a language specifically intended for meta-models.

#### *Use of the language to model itself*

For information modeling languages (and other modeling languages of which information is a part), it is possible to capture the concepts and many of the conceptual rules of the modeling language in the language itself. A model is itself a set of information; and that information can be modeled in the information modeling language. This approach has been used for the Object Role Model [99] and for EXPRESS [20].

This approach allows a practice sometimes called *introspection*: An information model can be used as a specification for a database (or some other information container); so the meta-model can be used as the specification for a database that contains models. A running software tool can access that database to determine the details of the model that is being used for some particular interaction, and use that information to formulate the proper messages and data structures for that interaction. This is clearly a useful concept for automating integration. But since the semantics of the meta-model is solely the semantics of the modeling language, the tool must also recognize elements of the database population – the names of objects, actions, or attributes – for the introspection to work.

### *Use of meta-modeling languages*

This alternative uses a language designed for the purpose of capturing the concepts of modeling languages, usually in terms of a very small set of "primitive" concepts, such as "thing" (noun), "action" (dynamic verb), "relationship" (static verb), "property" (adjective/adverb). The primary purpose of this approach is to produce a formal representation of the models that can be used for validation, for consistency testing, and for reliable translation into other modeling languages. In theory, this approach allows models written in different languages to be represented in a common language, with the consequence that models representing different views of the same system can be integrated into a consistent whole. And it may allow two models of the same thing to identify the common semantics, so as to facilitate interactions.

Three very different styles of meta-model have been developed using this approach:

The *axiomatic* approach defines the primitive concepts as undefined terms and characterizes them by axioms in a formal logic (predicate calculus) representation. It then defines the concepts of the modeling language via axioms involving the primitive concepts with the elements of a model (an instance of the modeling language) as variables. When the modeling language and its rules are axiomatized, any model written in the language can be similarly axiomatized by substitution rules. This approach allows the use of inferencing, theorem proving, and other artificial reasoning mechanisms for model analysis and mapping functions, such as checking model consistency, validating a population against a model, and verifying a translation of a model into another modeling language. In this approach, the mapping is not invertible. Model equivalence is based on the net semantic content of the two models – the set of axioms of one model that can be proved using the axioms of the other. This approach does not provide a good basis for testing the equivalence of model snippets (such as the part of the model that is relevant to a particular interaction), because equivalence is not based on individual constructs, and the proof of any given statement may depend on elements of the model that are not contained in the snippet (directly related to the interaction).

The *linguistic* approach, used in the Semantic Unification Meta-Model [102], defines the primitive concepts as grammatical elements of a natural language: noun, verb, adjective, adverb, and converts the modeling language to a grammar whose base elements are these pre-defined elements. Thus every model in the modeling language becomes a set of formalized sentences in the SUMM language. In theory, a parser that understands this language can understand any model written in any language, within the capabilities of the grammar. The problem is that most modeling languages have simple notations for constraints that become compound sentences and complex adverbial clauses when rendered into the SUMM language, and a parser that has a useful internal understanding of simple sentences may not have a useful understanding of such complex constructs. And as in the axiomatic approach, the mapping used in the linguistic approach is not invertible. But the linguistic approach can serve as a useful basis for simplifying models and comparing them. This approach can be used to determine whether two (simplified) model snippets are exactly the same, or to determine the set of sentences they have in common. If an interaction requirement is formulated in the same grammar, those capabilities may be sufficient to determine whether it is satisfied. [103]

The *hierarchical* approach used in the American National Standards Institute (ANSI) Information Resource Dictionary Services (IRDS) [85] and in the OMG Meta-Object Facility [83] creates a highest-level model composed of exactly the primitive concepts and the relationships among them. It then defines a (second-level) meta-model for a modeling language (or a set of modeling languages) as a *population* of the primitive-concept model. A model stated in any of those modeling languages is then stated as a population of the second-level meta-model. The Meta-Object Facility standard specifies the rules for converting a population of the second-level meta-model into a population of the highest-level meta-model. This approach allows model snippets from multiple languages with a common meta-model (as the case with UML, see 6.3.3) to be translated to and included in models for different

viewpoints and subsystems. It also allows languages with diverse second-level meta-models to be compared in terms of the highest-level meta-model, so that equivalent concepts can be identified. It is technically possible to compare models using languages with diverse second-level meta-models by converting them to their representations in the highest-level model, but the results are not generally useful unless the conceptual equivalents between the two second-level meta-models have been identified and one of the models has been converted to use the equivalents.

## 6.4 Semantic models = "ontologies"

The original definition of *ontology* from the philosophical context was "The science or study of being; that part of metaphysics which relates to the nature or essence of being or existence." [33] The term was given different usage and meaning by the artificial intelligence community, and in that community, it now refers to an artifact rather than a science or philosophy. There are now many different, but largely compatible, views of what an ontology is in the context of information technology:

- The semantic view: An ontology is the context needed to understand a specification, model, or other communication in the way that was intended. [15]
- The specification / reference view: "An ontology is an explicit specification of a conceptualization." [17] and "Commitment to a common ontology is a guarantee of consistency [in terminology]." [Ibid.] Simple taxonomies and thesauri are included in this definition as degenerate cases.
- The modeling view: An ontology is a metamodel. [3] (see 6.3.4)
- The automation view: An ontology is, or is captured in, a knowledge base designed to support automatic reasoning. [27] (see 6.5.1)

Combinations of these yield 11 more specializations of the root concept, all of which are valid. Our usage in this report is the intersection of the specification / reference view and the automation view.

Different representations of ontologies offer different kinds of rigor and support different kinds of automated analysis [112]. Some people use specialized languages for identifying concepts and the relationships among them, while others simply use UML notation for this purpose.

### 6.4.1 Knowledge representation tools and languages

With the exception of a few languages that were intended explicitly for exchanging knowledge between tools, most knowledge representation languages are defined by the toolkit that supports them. Consequently the capabilities of the languages and the capabilities of the tools are intimately related.

All knowledge representation tools create some kind of database, commonly called the *knowledge base*, containing the information that they capture. The features of the language the tool supports characterize the kind of information that can be captured in the knowledge base.

Another important property of the tools is the ability to perform *inferences*, also called *inferencing*. Inferencing is a built-in capability of the tool to use formal implications captured in the knowledge base to derive new "knowledge" information from the information that is directly entered into the knowledge base.

Knowledge representation languages, and the tools that support them, can be broadly categorized as frame-based, predicate logic, description logic, or hybrid.

#### *Frame-based representation*

The *frame-based* approach to knowledge representation uses "frames," which represent things, actions, and concepts (including types of things), and "slots," which represent relationships to other things, actions and concepts, including "properties" and their "values." [138] Frame-based models are very analogous to the object-oriented models and entity-attribute-relationship models described in 6.3.3, but they are much more relaxed in their rules for the content of a "slot," which give them more general expressive power.

In general, tools that support only frame-based knowledge representation provide only "database services for knowledge management" directly, along with some mechanism for invoking user-defined methods. They have little or no built-in inference capability.

### *Predicate logic representation*

*Predicate logic* representation is a machine-readable form of mathematical logic: It represents both facts and rules in the form of "well-formed-formulas" of the first-order functional calculus (also called the *predicate calculus*). The formulas involve logical variables, logical operators, universal and existential quantifiers (for all, there exists), "functions" and instance variables. A few functions are system-defined; the rest are named by the user, and may have natural language definitions, but are formally defined only by whatever formulas ("axioms") are stated to be true of them. While it allows for powerful inference, it can be a cumbersome knowledge representation.

The inferencing engine supports the three operations of the functional calculus:

- substitution — from: for any x <formula involving x>; conclude: <formula with y substituted for every occurrence of x> for any well-formed-formula y
- modus ponens — from: x implies y; and x; conclude: y
- modus tollens — from: x implies y; and not y; conclude: not x

Because substitution can generate a very large number of useless "facts," these systems require many control parameters to allow them to generate and retain only the conclusions that are useful to the application.

### *Description logic representation*

*Description logic* is a style of knowledge representation whose strong point is the ability to classify instances based on their attributes. The principal language structures are "facts" of the form:

<expression involving x>, where x is some specific object,

and "rules" of two basic forms:

- (for any x) if x is-a Y then <expression involving x> (is true)
- (for any x) if <expression involving x> then x is-a Z

where the expressions are essentially boolean expressions in programming languages (and most commonly LISP conditional expressions) and may involve the is-a operator. The interpretation of x is-a Y is that the object x is an instance of class Y.

Description logic tools are much more limited in what they can capture and deduce, but the representation has elegant theoretical and computational properties, and as a consequence, they are much more efficient and tractable than predicate logic tools.

### *Rule-based representations*

*Rule-based* languages are discussed in 6.3.1 as process specification languages, but the same style of language can be used for a kind of knowledge representation. The primary language structure is the "rule," of the form:

if <conditional expression> then <action>

where the action is a procedure call (in some closely coupled programming language) that may include the creation of new facts or rules.

The execution engine identifies the conditional expressions which are currently true and performs the corresponding actions. In a certain sense, such systems can perform more deductions than description logic systems, but the engine performs only *modus ponens* – it does no substitutions that are not explicitly performed by the <action> code provided by the engineer.

### *Hybrid representations*

*Hybrid* representations and toolkits incorporate features of description logic, frame-based and rule-based languages, and often other programming features. The purpose of the hybrids is to overcome the limitations of the "pure" representations.

There are many examples of tools and languages of each of these kinds. Corcho and Gómez-Pérez published a survey of supporting tools in 2000 [77]. Most of these represent academic projects in various stages of development. Evaluation of several such tools for use in the AMIS project (see section **Error! Reference source not found.**) is ongoing, and the results may appear in a future publication.

Resolving syntactic conflicts (see 5.1.2) and many semantic conflicts (see 5.2) requires some means of recognizing the equivalence of terms (or the lack thereof), either by common classifications and properties or by formal axiomatic definitions. Description logic systems may provide a mechanism for solving some such problems. Predicate logic systems can provide for the capture of axiomatic definitions, but the test for equivalence between two different axiomatic definitions requires the system to treat the axioms of one system as theorems and prove them using the axioms of the other. In general, this requires controlling the predicate logic engine in such a way as to produce a theorem-proving engine (see 6.5.2).

### 6.4.2 Semantic Web

The Semantic Web is a term coined by Tim Berners-Lee [73] to describe the realization of his vision of a next generation of the WorldWideWeb, which would supplement the data widely available in today's Web with "semantic markup" to allow smarter navigation and use of Web data. This vision also includes the development of agents that would understand and exploit this new markup to interact and perform tasks for human users.

The capabilities envisioned are:
- data discovery (see 6.10.1)
- question answering
- service discovery (see 6.10.2)
- service composition
- automated service interoperation

The Semantic Web idea has been embraced by a number of high-profile researchers, notably Jim Hendler (father of DAML, see below). Hendler emphasizes the "Web" aspect, envisioning that semantic markup and even the ontologies used for it will actually be a loosely connected network of subconstructs.

While the idea of loosely connected components works pretty well for people browsing compound documents, it creates several problems when applied to models used by autonomous software entities. Completeness, correctness, and consistency will not be achievable, and the long-term retention and locator validity issues that plague the Web today will plague the Semantic Web as well. As high volume and diversity of material characterize the Web, we may also expect them to apply to its semantic markup, and we are likely to encounter an even greater need to provide automation to help integrate these models.

A number of languages have been created and extended for the purpose of knowledge representation for the Semantic Web. Two main lines of effort in that area are the Resource Description Framework and the DAML family of languages. The other main area of effort is the semantic web services themselves. All of these are discussed below.

#### *Resource Description Framework (RDF)*

The Resource Description Framework (RDF) [130] is a standard structure for attaching semantic information to resources on the WorldWideWeb. A *resource* is any body of content that is accessible on the Web and identified by a Uniform Resource Identifier [128]. A resource can be a document, a part (element) of a document, an image, etc. The RDF specification is divided into three parts:
- RDF Model and Syntax
- RDF Schema
- RDF Model Theory

The project has extended the knowledge representation capabilities by inclusion of the Ontology Inference Layer (OIL).

The RDF Model and Syntax [131] is based on a triplet consisting of *subject*, *verb*, and *object*. The "subject" can be any resource. The "object" can be a resource or a *value* (a simple string) or nil. The "verb" is a string that represents a predicate, a property, or a relationship, possibly coupled with a reference to a resource that defines the verb formally. A triplet can specify a relationship between two resources, attach property information to a resource, or attach constraints or behaviors to a resource. The triplet itself can be a resource, which allows assigning properties, relationships and constraints to properties and relationships. An RDF model can be graphically represented by a labeled directed graph, which allows the RDF syntax to be a representation of a "conceptual graph" [129][137].

The RDF Schema [132] defines terms (verbs and values) for commonly used semantics, such as "subClassOf," "subPropertyOf," "domain," and "range." The "subClassOf" verb implies the parent/child relationship of any two classes, that is, every resource that is a member of the child class is also a member of the parent class. The "subPropertyOf" implies the parent/child relationship between two properties. The "domain" and "range" properties constrain the admissible subjects and admissible objects for a given verb.

The Model Theory [133] is a formal definition of the semantics of the RDF schema and model elements, for use by a reasoning engine (see 6.4.1).

The Ontology Inference Layer (OIL) [134] extends the RDF model with description logic (see 6.4.1). OIL allows the Model Theory to be extended by formal specifications that permit a class to be defined by a set of properties, such that every resource that has those properties can be determined to be an instance of that class. It also supports formal specifications for the derivation of the value of a given property from other properties and relationships, which may be information modeled as RDF properties of the resource, or of related resources, or, in some cases, values derived from the content of a related resource.

### *DAML Language Family*

The Defense Advanced Research Projects Agency (DARPA) Agent Markup Language (DAML) [104] is not a single language but rather a family of languages for defining machine-interpretable models for a semantic web. The members of this family that relate to this discussion are:

- DAML-ONT [136] – Original DARPA Agent Markup Language, a description logic language
- DAML+OIL [135] – Replaced DAML-ONT. Combined features of DAML and the Ontology Inference Layer (see above)
- DAML-L – Proposed logic language to supplement DAML+OIL which would allow the expression of Horn clauses (a subset of first-order predicate logic)
- DAML-S [75] – Originally to be a Web service ontology expressed in DAML-ONT. But the project extended DAML-ONT to meet this objective, and thus defined another DAML dialect
- DAML-Rules – A proposed means for adding rule-based language features to DAML, currently in early development

That is, there is a DAML dialect, or a project to create one, for every common form of knowledge representation. Unlike the knowledge representations described in 6.4.1, however, the DAML languages were not developed for particular knowledge engineering toolkits. Rather they were developed as a common means of knowledge representation that could be processed, whole or in part, by several different kinds of tools.

The WorldWideWeb Consortium currently has an effort underway to define a Web Ontology Language (OWL), whose intent is in part to unite the RDF and DAML work in knowledge representation. OWL is likely to be a further evolution of DAML+OIL.

### *Semantic Web Services*

A number of tools and languages, can be employed to support dynamic discovery and composition of web services, which in turn allows the execution of certain tasks to be automated [74]. The tools include knowledge

representation languages (6.4.1) and inferencing systems (6.5.1), resource discovery (6.9.5), planning algorithms (6.7.1), interface languages (6.3.2) and middleware (6.11.1). But many specific markup collections using these languages are also required, including:

- Descriptions of the capability and other properties of the webservices in a standard semantic markup language
- Web service ontologies — dictionaries of formal definitions of common terms and terms used in specific disciplines or regions — to form a basis for interpretation of a given webservice description
- Partial models of processes making use of web services — templates for webservice composition
- Markup of user and organization preferences and policies, to form a basis for selection

The automation envisioned is:

- Automatic Web service discovery — finding a service that fulfills the requirements of a service request
- Automatic Web service execution — an agent executes a web service on behalf of a client
- Automatic Web service composition — "the automatic selection, composition, and interoperation of appropriate Web services to perform some task, given a high-level description of the task's objective" [74]

The Web service markup is envisioned as being layered on top of WSDL [79] specifications for Web services (see 6.3.2). Markup done to support the prototype in [74] was written in a combination of DAML+OIL (see above) and a subset of first-order predicate logic.

The automated functions identified for "semantic web services" are very similar to those required for automation of integration tasks, as discussed in sections 3.4, 3.5, and 3.6. In a similar way, the technologies which may be exploited to perform the semantic web services automation are among those discussed in Section 6 with respect to their use in automated integration. It follows that much of the further work of the Semantic Web Services projects may be of immediate value to the AMIS project.

### 6.4.3 Lightweight ontologies

A "formal ontology" is an attempt to articulate concepts of interest to a certain locus of usage [140] in a way that is directly processable by particular software technologies for "reasoning" (see 6.4.1). The locus of usage might be a community, an organization, a set of applications, or a set of processes. Conversely, any attempt to articulate concepts of interest for a locus of usage, in such a way that the concepts can be reused in contexts different from that in which they originally appear, may be considered an ontology.

Once we take this view, we can find ontological information in many forms other than the formal knowledge structures explicitly labeled "ontologies". Some of the likely sources are domain models (including the "paper" models found in textbooks on particular disciplines), taxonomies and terminology dictionaries for particular domains, software system specifications, application programs, scripting languages, and database schemas. The degree of explicitness varies significantly among these types of sources. A domain model may be so carefully and completely stated as to represent a formal ontology but for its form, which is often a combination of graphics, natural language and mathematical or scientific formulae. An object-oricnted system specification will explicitly define "object types", and thus concepts, although they may not be so specified as to promote general reuse [141]. An application program may not articulate its underlying ontology at all, and yet there is inevitably an implicit structure of concepts that are represented in and enacted by the program text. Even the tacit processes enacted by members of an organization, including conventions of person-to-person communication, are underlain by implicit ontologies [142]. And both explicit and implicit rules for communication within an organization involve the use of a large body of natural language concepts with conventional interpretation. This implies that common dictionaries and thesauri are sources of ontological information as well.

Interoperability is fostered when we can identify such *lightweight ontologies* and exploit them for what they really are: informal, partial or tentative attempts at standardization of concepts and terminology. The easier it is to extend the locus of applicability of the implicit ontologies of software systems and business processes, the easier it will be to integrate disparate applications. And therefore, the integration process may benefit from approaches to the capture and intepretation of "informal" and "lightweight" ontologies. These include:

- software specifications (see 6.3)
- machine-readable forms of dictionaries and thesauri, such as WordNet [145]

- manual efforts and specialized software for extracting concept graphs and behavior rules from text (but the integration process uses only the formalized results)

Moreover, we should not expect the exploitation of lightweight ontologies to be a one-pass process. The organization of information for a particular software process creates a new ontology, and therefore, the process of integration creates a new ontology. And that new ontology is the foundation for the next level of software development, and the foundation for the next level of concept standardization. A similar effect can be observed in business processes that become standardized and business processes that become automated. As we do more, the conceptual substrate evolves [143]. And in every case, the new ontological information will initially be captured in the software and in the business communications, not in formal ontological models.

### 6.4.4 Ontology-based approaches to integration

There are several ontology-based approaches to integration. Gruninger and Uschold [146] provide the following categorization:

#### *The interlingua approach*

The interlingua approach assumes that there is an axiomatized reference ontology that covers the scope of the interactions, that each interacting agent has a (prior) complete mapping from its internal ontology to the reference ontology, and that each such mapping preserves the axiomatic interpretation of the reference ontology. This allows the automated derivation of a mapping between the two agent ontologies via inference engine. Note: this is analogous to the "exchange standard" approach, which assumes a complete mapping of the agent's internal information objects to/from the standard exchange objects, but for most exchange standards the exchange model is not axiomatized and the mappings are formalized only in executable code; so there is no way to validate the preservation of semantics.

#### *The common ontologies approach*

The common ontologies approach assumes that each agent has a formalized ontological model of the objects and behaviors that appear at its interfaces, and that most of that model comprises, or is mapped to, elements of a publicly available ("common") ontological library, although the agent may have its own local extensions. It also assumes (unlike the interlingua approach) that there are *multiple* common libraries that have overlaps in their semantic content and have partial reference translations to each other. For two interacting agents, this allows automatic generation of a verifiable mapping between those elements of their ontologies that are part of the overlap in the common ontologies they use. In the best of cases, this mapping covers the transactions of interest. Otherwise it reduces the set of ontological elements for which heuristic mappings (see below) need to be developed, and provides a common basis for that development.

#### *Heuristic approaches*

Heuristic approaches deal with those situations in which there is insufficient formal basis for a verifiable mapping between the ontologies of two agents that covers their interactions — incomplete reference ontologies or incomplete translations (both agent ontologies contain local extensions used in the interaction), and non-conformant agent ontologies (conflicting agent/reference models, mappings that do not preserve all axioms). They involve generation of hypothetical mappings between the agent ontologies and the use of search strategies and evaluators to choose a mapping most likely to produce successful interactions. They use negotiation protocols to discover common ontological elements and steer searches and evaluations. In the best of cases, they are based on verifiable mappings for some part of the two ontologies and are limited to resolving conflicts or finding equivalences among the unmapped local extensions.

Mitra et al. [30] describe a semi-automatic tool that finds similar concepts in different ontologies based on labels, context identifier tags, term-based matching rules, structure-based matching rules, and other rules.

#### *The manual translation approach*

The manual approach involves development of a specific mapping from the relevant elements of the ontology of agent X to the corresponding elements of agent Y and the reverse mapping, for each pair of agents (X,Y) that will interact. This development uses a combination of formal methods and human expertise, and the result cannot be validated if either agent ontology is not axiomatized.

Calvanese et al. [5] propose a query or database view-like approach for mapping between ontologies. This approach enables formal mappings to be constructed in cases where there is no direct correspondence of terms. The qualities of such mappings, such as the conditions under which they can be used bidirectionally, can be postulated by analogy to the view-update problem in database literature [8].

## 6.5 Artificial intelligence algorithmic methods

### 6.5.1 Knowledge engineering and expert systems

*Knowledge engineering systems* consist of a database system, which captures not only "facts" (information in the conventional database sense) but also "rules" for the derivation of new facts from those directly entered, and an "inferencing" or "rule execution" engine to perform the derivations. Most knowledge engineering systems are intended as toolkits with which to build expert systems. There is typically a close relationship between the capabilities of the system and the style of knowledge representation (see 6.4.1) it uses.

All of these systems distinguish between the "knowledge base" – the facts and rules retained for use in all future applications – and the "scenario" – the set of "hypotheses" for a given application. The hypotheses have the same structure as "facts" but they are only assumed to be true for this application, and the deductions made by applying the facts and rules of the knowledge base have the same property.

*Expert systems* are automated advisors that suggest appropriate actions to business and engineering practitioners, and in some cases create the necessary data files and transactions to perform those actions. (The currently most widely used expert systems are software configuration "wizards.") Expert systems use question-and-answer to advise a human agent, working from a populated knowledge base and an inferencing mechanism that determines the implication of the answers for both the results to be produced and the next question to ask, until a final recommendation or result is developed. Although many expert systems are built on knowledge-engineering systems, many others use hard-coded inference rules and embedded facts or private databases.

A number of technical integration tasks are, for the most part, rote application of particular patterns with problem-specific substitutions and a few other design choices. Expert systems can be readily employed in developing tools to perform such technical integration tasks, or to assist human engineers in performing them.

### 6.5.2 Automated theorem proving

Theorem-proving tools search for sequences of rule applications to show that a targeted hypothesis follows logically from axioms and previously proven theorems. Many different theorem-proving tools are available, many of them implementing interesting and unique approaches. However, there exists a mature body of work that, along with the related topic of logic programming, is part of the average computer science curriculum.

Duffy [44] describes the theory and limitations of how automated theorem proving relates to "program synthesis" from "algebraic specifications." Processing formal specifications of components to determine how to construct a desired system is an extension of that idea.

As stated in 6.4.1, proving that two terms with axiomatic definitions have the same semantics requires the axioms of each to be proved using the axioms of the other (and any underlying ontology). This is a direct application of theorem-proving technology.

### 6.5.3 Declarative programming

Declarative programming languages such as Prolog incorporate features of theorem-proving and possibly planning systems to relieve the programmer of the burden of specifying precisely how a desired behavior will be accomplished. They may be useful to assist in the implementation of self-integration.

## 6.6 Machine learning and emergent behavior

This section addresses technologies that allow components and systems to modify their own behaviors. These technologies may be useful for self-integrating components (see 2.4) and for resolving embedding conflicts (see 5.3.3).

### 6.6.1 Neural nets

Neural nets [28] are based on the internal behavior of the human brain. The software version of the neural net model involves nodes that have responsibility for some problem-specific information units, and have adjustable internal thresholds or tests that will cause them to "fire" or transmit a unit message to some set of receptors on other nodes. The receptors on a node react to certain unit messages. Some receptors alter the problem-specific data, while others alter the firing thresholds. Some receptors react only to internal transmissions; others react to external information and events. The system is repeatedly exposed to examples from the problem space and to feedback on the correctness or goodness of its reactions, until the neural net has "learned" to respond "appropriately" to the examples. Thereafter, it can be used to exercise that learned behavior in "production" instances.

Neural nets tend to be most useful in solving pattern recognition problems and in control problems in which damping is needed.

### 6.6.2 Interacting agents

The term "agent architecture" actually includes three kinds of software systems that have a few features in common:

- systems built on an "agent-oriented programming" platform – a set of components that communicate using a blackboard communications technology with a somewhat extensible collection of participants, without regard to the complexity or specialization of the components
- *personal agents* – systems that incorporate Web-search or data monitoring techniques and automated decision-making strategies to perform low-level actions that would otherwise require human attention, and usually to alert a human agent when some important event has occurred
- *autonomous agents* – systems built from multiple groups of nearly identical components with independent rule-based decision-making strategies, each of which maintains the information about a different (typically real-world) object, publishes some part of that information, and has access to the publications of some or all of the other components, usually using a blackboard communications technology

While each of the first two incorporates one of the concepts – blackboard with an extensible component list, or rule-based decision-making – it is the last, which combines the two with the "identical and independent rule-base" concept, that produces "emergent behavior." Autonomous agents compete, collaborate and interact in somewhat unpredictable ways to find solutions for which there is no deterministic algorithm. [105]

Autonomous agents perform well in solving problems that require team solutions and dynamic coordination. It may well be possible to formulate problems of mapping resources to roles, or data availabilities to data requirements, in such a way that they can be solved by gangs of autonomous agents, but it is likely that each such agent would have to possess considerable knowledge bases and semantic reasoning capabilities. And in most cases, a single decision-making tool with those same facilities could reach the same result with a deterministic algorithm.

### 6.6.3 Adaptive and self-organizing systems

Adaptive systems are systems that have a number of pre-programmed or scripted behaviors whose activation and parameters are controlled by the outputs of functions that analyze events and states in their environment. The behaviors of such systems therefore change according to what is happening around them – they react. [88]

Self-organizing systems are similar, but the activation of their behaviors and the parameters of those behaviors are controlled by weights. These weights are adjusted by the functions they use to analyze the environment and by some of their own behaviors. The result is that the past experience of the system, together with the current situation, conditions its responses.

Both of these techniques may be useful in building components that can achieve interoperability by adjusting some of their behaviors to meet the needs of particular communicating partners.

In restricted contexts, it might be possible to "train" a self-organizing agent to perform a desired behavior without programming it explicitly. If an agent is capable of recognizing the desired behavior (or a means to that end) when it observes that behavior, an economy of incentives and disincentives among agents could suffice to produce the desired behavior out of a sort of behavioral conditioning, even if no logical plan for achieving it was formulated. This differs from any other emergent behavior only in the presumption that self-integration ("learning" to cooperate in new ways) is a prerequisite.

## 6.7 Industrial engineering methods

The industrial engineering methods of interest are those directed to finding solutions to problems that nominally involve evaluating large numbers of complex alternatives. Many of these methods may also be considered operations research approaches, and in computer science they are commonly called "search strategies."

### 6.7.1 Planning algorithms

Planning, as described in the literature, is properly a class of problem, for which there are many diverse approaches to solution. [113] [114] Planning problems can be characterized as having a *goal* – a result to be obtained, an *initial task network* – a set of activities to be performed, a set of *operators*, and a set of *method templates*. The *goal* can be stated as a condition on the state of the system – a boolean expression involving variables in the system state space – or as minimizing/maximizing the value of an *objective function* whose parameters are variables in the system state space. The *initial task network* specifies a set of complex tasks that must be performed, possibly with partial ordering and other constraints. The *operators* represent the available "primitive tasks" – pre-defined activities by system resources – by specifying the pre-conditions for initiating them, their effects on the system state while being performed, and their effects on the system state when completed. (Their effects while running are only important when time is a part of the characterization.) An operator is usually stated as a paraform with a set of parameters whose values are substituted into the pre-conditions and effects on the system state. A *method template* specifies how a complex task may be accomplished as a sequence (or more complex network) of operators or other complex tasks, including additional pre-conditions, running effects and final effects that are consequences of the method template (or the complex task) and go beyond those implied by the operators. For every complex task (the initial tasks and any intermediate ones mentioned in method templates) there is a set of one or more method templates, specifying alternative *decompositions* of that task. Like the operators, the complex tasks are usually parameterized, and the parameter values are substituted into the parameters of the operators, pre-conditions and effects appearing in the method templates. [115]

Conceptually, the solution to a planning problem involves generating all of the possible decompositions of the initial task network as networks of operators with all parameters specified, and then choosing that decomposition which best accomplishes the goal. But in most cases, the search space of all possible decompositions is not computationally tractable, and planning algorithms are distinguished by their mechanisms for steering the search and eliminating the creation of decompositions which are unlikely to be solutions. When the goal is minimization/maximization of the value of the objective function, many planning algorithms are designed to

produce estimates of the difference between the "current best value" and the "true best value" and to stop the search when the result is good enough, but not necessarily best.

Unless they are strictly rule-based, planning algorithms must use some form of intelligent search to reduce a combinatoric space of tasks, orderings, and resource assignments to a small set of desirable choices. Similar search strategies may be useful for finding possible paths from the available behaviors to the desired behavior.

The specification of coordination models may be viewed as a planning problem. As a planning problem, the need for an information flow, interface descriptions, and the description of the required interactions derived from joint activites in the system provides an initial task network. The details of the protocol (e.g. synchronous/asynchronous, message sequencing, maximum message size, etc.) provide information needed to specify method templates and operators.

Planning strategies may also be used for finding possible paths from the available component behaviors to a desired functional behavior of the system. Different paths may include different speculations about what resources or capabilities that are not already available could be created or obtained along the way.

### 6.7.2 Mathematical programming methods

Optimization algorithms and improvement methods are used to find a set of inputs that causes a specified function to yield a desirable output when a direct, analytical solution is not available. The mathematical methods focus on maximizing or minimizing the value of an objective function under a set of "constraints," which are mathematical formulations of the limits on the possible values of the variables involved in the solution. There are many such algorithms, which differ primarily in the mathematical nature of the constraint equations and objective functions they can support, and thus in the kinds of problems to which they are most successfully applied.

Such methods dominate industrial engineering and economic planning activities, but it is not clear that they have any immediate application to the problems of automating integration. In most integration problems, the primary objective is to find a feasible solution, where most of the constraints have the form X = Y, and X and Y do not have numeric representations..

### 6.7.3 Genetic algorithms

Genetic algorithms use a search strategy that alternates between expanding a set of tentative or partial solutions in one or several ways (mutation, combination, or random generation) and contracting the set by eliminating the least promising ones. Combination mates two promising candidates to produce one or more "child" candidates having combinations of features (algorithmically or randomly selected) from both parent candidates. Mutation alters a promising candidate by (usually randomly) changing one of its features to a different allowable value. Random generation creates a new candidate by algorithmically selecting feature values out of the possible value set, usually randomly with some algorithmic controls.

Genetic algorithms represent one possible approach to the path planning problems identified above, because they are capable of dealing with discrete non-quantitative feature values, and with constraints on the relationships among such features. Genetic algorithms also suggest that one could implement a capability to speculate about resources and capabilities that are not already known, by mutation or recombination of known resources and capabilities.

## 6.8 Data systems engineering

### 6.8.1 Model integration

Model integration here refers to the integration of data models, information models, conceptual models, and/or structural models in a software context. Although much of the work labeled "schema integration" or "ontology integration" is equivalent to or interrelated with the task of model integration, it is difficult to find relevant references to model integration by that name, at this time.

Quite a lot of references exist for *enterprise* model integration. Most enterprise models are not the kind of models this paper addresses. However, the approach described by Fillion et al. [14] is applicable to many kinds of models. Modeling languages are characterized using ontologies, effectively creating a new, unifying metamodel. This addresses the "language barrier" that exists when integrating models that are written in different modeling languages and supports formal reasoning about the structural content of models. Other information needed to support reasoning about the integration of different models can then be captured in "context" ontologies.

For models that already share EXPRESS [20] as a common language, the Visual and Natural Language Specification Tool (VINST) [6][7] judges the similarity of different entities using the corpus of information that is available in the context of ISO 10303, a collection of standard models for the exchange of product data [19]. This includes full text definitions of terms in addition to the labels and structural information in EXPRESS models.

For models that already share UML as a common language, Rational Software Corporation's Rational Rose [37] includes a tool called Model Integrator that allows several UML models to be merged into one. However, automated integration of "unrelated" models (in the sense "not derived from a common ancestor") was not in the original design intent of the tool [36], so its functionality in that regard has not yet been developed to the level of sophistication seen in existing schema integration methods.

### 6.8.2 Schema integration

Research in database schema integration that was *called* database schema integration peaked in the 1980s. However, as attention shifted to federated and heterogeneous information systems and to ontology-based research during the 1990s, much of that research continued to relate to schema integration (or vice-versa, depending on one's viewpoint). The fundamental problems of detecting semantic overlap and heterogeneity among different information sources were essential to all of this research, and they remain essential to our present topic.

Thousands of papers about schema integration were published. The most one can hope to do is cite a reasonable set of representative works. The selections cited here were found by breadth-first exploration of the references made by several convenient sources. Thus, the citation of any particular work is indicative only of its reference proximity to the original sources and should not be construed as an endorsement of one research program over another.

Batini, Lenzerini and Navathe published a survey of integration methodologies that were available circa 1986. By that time, the various kinds of conflicts that can arise during schema integration had been analyzed in various ways, and it was acknowledged that "automatic conflict resolution is generally not feasible: close interaction with designers and users is required before compromises can be achieved in any real-life integration activity." [2]

Of particular interest to the task at hand are any automated or potentially automatable approaches to finding similar entities in different schemas. The survey mentions two references in which "the integration system automatically assigns a 'degree of similarity' to pairs of objects, based on several matching criteria." In the first [1], these criteria appear to include similarities in the attributes and relationships of entities and in the attributes and involved entities of relationships in an Entity-Relationship model. However, the heuristic is not described in sufficient detail to reproduce it exactly. In the second [9], a measure of the degree of similarity is calculated based on detailed, formal assertions supplied by the database designer. A work published later by the same group of researchers [26], provides a more detailed analysis of attribute equivalence. Their prior analyses of object class equivalence [10] and relationship equivalence [31] are then revisited from the perspective of attribute equivalence. Similar work based on a further refined definition of attribute equivalence was reported by Sheth et al. [38]. The automatable portions of these analyses depend on assertions that involve real-world state, "the underlying real world instances of the object being represented," or a similar notion for real-world attribute semantics. The rigor of the analyses therefore depends on the rigor with which it can be determined that two different abstractions refer to the same real-world object, which is another form of the hard semantic-matching problem. As Reference [38] cautions, "Schema integration involves a subjective activity which relies heavily on the knowledge and intuition of the user about the application domain (domain of discourse), intended use of the integrated schema, and the systems that manage data modeled by the schemas. Thus, the results of a schema integration activity are not unique and *cannot* be generated totally automatically."

References [22], [23], and [39] document a line of work that deals with a related notion of "semantic proximity" between different database objects. Reference [23] begins with the assertion, "The fundamental question in interoperability is that of identifying objects in different databases that are semantically related, and then resolving the schematic differences among semantically related objects." Semantic proximity is defined as a function of context, abstraction, domain (values), and state (extents). The line of work includes a breakdown of schematic heterogeneities into domain incompatibilities, entity definition incompatibilities, data value incompatibilities, abstraction level incompatibilities, and schematic discrepancies, each of which is further broken down into different kinds of conflicts. Different taxonomies of schematic heterogeneities and resolution methods appear in many other works about schema integration and federated, distributed, heterogeneous, and multi-database systems ([4], [25] and others).

The database schema integration work that survived through the 1990s became increasingly mingled with research having to do with ontologies, as in Reference [24]. Progress in this direction can be seen in Reference [18], where a "semantic dictionary" is used in the detection and resolution of semantic heterogeneity. Similarly, Reference [16] describes a semi-automatic tool for finding similar classes in "semantically enriched" relational schemas. However, this semantic enrichment corresponds more closely to the information contained in a UML class diagram. Their approach to finding similar classes focuses on generalization/specialization and aggregation, ignoring the attributes. Reference [11] describes an approach to finding similar classes based on reasoning about fuzzy terminological relationships between names as defined in a terminological knowledge base. Other progress in the ontology direction can be seen in Reference [40], where it is proposed to replace the assertion-based characterization of attribute semantics used in previously described work with a characterization in terms of a "concept hierarchy." With respect to automation, Reference [18] repeats the assertion that "automatic conflict resolution is in general infeasible" and Reference [16] makes a related assertion to the effect that detecting semantic relationships among different databases cannot be completely automated, whereas Reference [40] proposes to semi-automate the identification of concepts using natural language processing of a supplied data dictionary.

Related problems in information retrieval continued to be addressed ([34] and others) until this line of work was transformed by the emergence of the World Wide Web.

### 6.8.3 Enterprise Application Integration

*Enterprise application integration* (EAI) is a category of software tools whose primary function is to support business processes by linking up distinct software systems in the overall enterprise computing environment. This is a relatively new and fragmented class of software, employing a number of techniques of varying sophistication and providing quite variable capabilities. Gold-Bernstein [86] describes them as "a selection of technologies to address a number of applications integration problems."

EAI technologies include:

- platform integration solutions – "object request broker" middleware, messaging, or publish-and-subscribe technologies (see 6.11)
- message brokers, that perform intelligent routing of messages between applications, and may do translations and transformations
- process automation and workflow

The EAI package also includes setup tools that allow the system engineer to define the processes and the routing and mapping rules.

The critical element of the EAI package is a set of application-specific "adapters," each of which is capable of providing the technical interfaces that integrate a particular application component into the EAI system. The adapter activates functions of the application, extracts data from and provides data to that application software package in some form convenient to that application, using the interfaces that the application itself provides. This may include direct data update/retrieval via APIs, or simply requesting the application to read or write a file or update a database.

Most EAI systems include a central workflow engine that orchestrates the system processes and activates resources and data flows as needed. The central engine communicates with the adapters to effect the activations and data flows. In a few cases, the adapters are specialized agents in an agent-oriented architecture, each of which responds

to particular stimuli in a publish-and-subscribe environment by activating functions of the application and obtaining or providing data in commonly accessible areas.

In some cases, the adapter is responsible for converting the application information to/from a common interchange model and form used by the EAI package. In those cases, the preservation of semantics between the application and the system is the responsibility of the adapter. In others, each adapter obtains and produces information in a form convenient to the application, and the central engine is responsible for the conversion of the messages or files between the adapters, using a common reference model or an application-to-application-specific translation, or both. In those cases, the adapters have only the application semantics, and the preservation of the system semantics is the responsibility of the central engine. In some cases, the reference representation is a standard that at least some of the systems involved can use. In other cases, the reference representation is the favorite representation of the "dominant application" and the translators map to it.

Most existing EAI systems offer some tooling that supports the systems engineer in identifying the data flows and transformations that are needed for his business process, and captures the rules the EAI system needs to implement those flows – moving, converting and deriving all the needed data for each interaction. The rules tell the system when to activate, what to activate, what to provide, what to convert, what conversion tool to use, etc. As a consequence, the quality of the results is primarily a result of:

- the knowledge and understanding of the human engineer(s), and
- the power of the process/rules language used by the EAI system (and the engine or agents that execute it).

The primary contribution of EAI systems to the enterprise is in standardizing the communications technologies for the enterprise system and providing an automated mechanism that implements the process and data flows. But the functional integration is defined completely by the systems engineer, and all of the semantic and technical integration concerns are addressed in developing the adapters or the central engine. The diverse ideas for the integration architecture that are embodied in EAI systems, however, are all worth considering.

## 6.9 Negotiation protocols

Several protocols have been developed and standardized to allow agents to negotiate the conventions that will be used for their interaction. In all of the standards, the assumption is made that there exists a (potentially large) set of published conventions, and each agent has been constructed so as to be able to support some set of those. The object of the negotiation is to find a common convention that both agents have been constructed to support.

There are several active research efforts directed toward dynamic negotiation of a common interpretation model of the terms to be used in an interface for which the basic communications conventions have already been established (by whatever means). Those mechanisms are based on reference to a standard or registered ontology, which is analogous to the mechanisms discussed below. But some of that work supports an additional feature: Each agent provides a formal specification of its own "extensions" to the reference ontology, and further negotiation allows the communicating agents to determine which of those extensions have a joint interpretation. These approaches are discussed in 6.4.3.

### 6.9.1 Standard interfaces

Standard interfaces is the time-honored solution. The negotiation is done off-line (usually in committee) and the software component is built to support one or more such standards. The systems engineer configures the component to use one of the standards it supports for all of its interactions of a given kind. There is no online negotiation.

There are two basic forms of interface standards:

A standard exchange form defines the format for a file or document that contains a particular kind of information set – the nature of the information units, and their organization and representation – and the component is built to be able to read or write that file/document format.

A set of standard operation signatures, or *application program interface* (API), defines the names for a set of functions that the component can perform, and the parameters to those functions. To be useful in a distributed environment, the API must be specified in an interface definition language (see 6.3.2) that is supported by a set of standard communications protocols.

### 6.9.2 ebXML Collaboration Protocol Agreement

In an effort to segregate the problems of defining standards for Web-based business interactions, the ebXML consortium developed a framework [96] using the following concepts (among others):

- The Collaboration Protocol Profile (CPP) is a statement of the ebXML capabilities of a party. It is a description of a set of messages, each of which has a name, a set of required information units and a set of optional information units. Each message and information unit has a stated meaning, purpose and form. The form of units and messages is defined using XML languages; the meaning and purpose is defined in natural language text.

- The Registry is a repository of ebXML documents, accessible over the Web, where parties may register CPPs, and in some cases, servers and the Profiles they can support. Initially, there is a canonical registry (maintained by OASIS), but conceptually there may be more than one such registry.

- The Party Discovery Process involves a set of standard messages that all ebXML agents are expected to support. One of the messages opens a conversation, identifies the calling party and the called party, and offers a list of registered Collaboration Protocols that the caller can support, with indications for preferences. One response message accepts one Collaboration Protocol from the list offered and offers to conduct a business conversation (and possibly subsequent conversations) using that Collaboration Protocol, with certain role assignments and possibly certain options. The confirmation message sent by the caller establishes a Collaboration Protocol Agreement for the conversation(s) by agreeing to use the specified Collaboration Protocol in the way proposed.

- The Collaboration Protocol Agreement (CPA) is the agreed-upon set of rules for a given conversation that are derived from the CPPs, the assigned roles of the parties, and the chosen options.

The expectation is that a given agent will have been pre-programmed to be able to support a particular list of registered Collaboration Protocols, and in many cases will make the determination on the basis of a match on the name of the registry and the name of the Collaboration Protocol. A more intelligent agent may access the registry, obtain the text of the protocol specification and compare it to the texts of the protocol specifications it understands, and accept what appears to be a known protocol under another name. (This may be necessary if more than one registry is commonly used in Web-based commerce.) And an intelligent agent may use artificial intelligence approaches to determine semantic equivalence of protocol specifications that do not have identical text, and possibly adjust its internal configuration to support a strange Collaboration Protocol for which it can make strong semantic correlations.

### 6.9.3 OMG Negotiation Facility

The Object Management Group Negotiation Facility specification [82] defines a protocol which is similar to the ebXML CPA (see above) in every regard but one: It does not presume the existence of a "registry" per se. It assumes that every potential protocol for the conversation will have a globally unique identifier of the form (source identifier, protocol identifier), but the text of the protocol specification may or may not be accessible online. The "source identifier" is required to be a unique "object" identifier using one of several standards, including but not limited to a Universal Resource Locator (URL) that would be the identifier for an ebXML registry.

The OMG Negotiation Facility is formulated as an "object service protocol" (rather than a set of messages) using the ISO interface definition language (see 6.3.2), and it seems to presume that the selected protocol for the conversation will be of the same kind. (The expectation is that the same *underlying* technology and protocols will be used for both the negotiation and the conversation.)

### 6.9.4 Contract Net

Contract Net is a protocol that supports the economic model of negotiation. The ideas for the Contract Net protocol began with the consideration of "negotiation as a metaphor for distributed problem solving" as a research topic [43]. Today, the usable specification of the protocol appears in the Foundation for Intelligent Physical Agents (FIPA) Contract Net Interaction Protocol Specification [45], and reusable implementations of the protocol are provided as an integral part of some agent building toolkits. (e.g., [89]) Using such a toolkit, one can attempt an agent-based solution of any problem that can be formulated in economic terms. Since the implementation of the protocol contains generic problem-solving logic, the experience is similar to declarative or logic programming (see Section 6.5.3).

### 6.9.5 Ontology negotiation

Exploiting lightweight ontologies (see 6.4.3) to enable communication between agents involves articulating each agent's ontology and then mediating them to achieve a workable common ontology for their interactions. The less formal the articulation — i.e., the weaker the semantics — the easier it is to articulate, but the more difficult to mediate. For example, the articulation may be a simple listing the terms and concepts that seem to be relevant to the interaction; but there is no obvious approach to determining whether the agents' concepts are identical or to resolving differences in terminology. *Ontology negotiation* [144] is an approach to mediating ontologies when little can be assumed about the depth and formality of the semantics each agent possesses. It is an automated "runtime" process implemented as a conversation between the agents. The negotiation process is modeled on processes in human-to-human communication, in which terminology differences and conceptual gaps are bridged through iterative and recursive interpretations and clarifications. The underlying assumption is that useful mediation does not necessarily require translating complete ontologies, and that the agents involved will be able to decide when enough mutual understanding (an adequate common ontology) has been developed for the intended interaction to proceed.

There are several experimental protocols for ontology negotiation, all of which are similar in design. All of these protocols assume that both agents have a common initial understanding of the nature of the interaction and their roles in it. The subject of the negotiation is the objects to be operated on and perhaps some special properties of the actions to be taken. Most of the experimental work has been done with client agents attempting to obtain useful information from an information repository front-ended by a strange server agent. In those cases, there is *a priori* agreement on the roles and the sole operation – find information about X – and on the use of the negotiation protocol. What is being negotiated is what the client means by X, where X is a subject with some constraints. While this kind of integration problem is not common in the development of integrated "intramural" systems, it will become increasingly common in the future, when the "available resources" (see 3.3) include information repositories accessible over the Internet.

## 6.10 Resource identification and brokering

This section deals with technologies for locating active resources that can provide a certain set of functionalities, or passive resources that provide the necessary information base for the execution of certain functions.

### 6.10.1 Resource discovery

Resource discovery is the science of locating resources (or services) whose sources and/or availability are not already known. The resource discovery *problem*, i.e., the *inability* to locate useful resources on the Internet, was already a research topic in 1988 [48][47] and is still being worked on today [41][50].

Resource discovery has many aspects:

- Request and resource description disambiguation
- Heterogeneous information integration
- Classification, cataloging, and indexing
- Request routing, brokering, and/or mediation
- Searching

- Navigation
- Collection, maintenance, and expiration of information from non-cooperating resources ("web crawling")
- Information retrieval

A resource discovery capability is needed if an integration tool is to be able to include outside resources and services as part of its plans to achieve integration.

### 6.10.2 Service and protocol brokers

A *service broker* is a software system whose primary function is to answer the request: Find me a resource who can do X. X may be a pure information technology service, or a business service, or the provision of a product.

While the fundamental elements of the technology are the same in all cases, there is a significant difference between brokers whose function is to generate revenue for the provider of the broker and brokers whose function is to provide a pure information technology service. Many concerns of the former class are related to protecting their market and charging for their services, and those concerns are entirely absent in the latter class.

For the pure information technology broker, there are five important concepts:

- The broker has an internal knowledge base that associates "resources" with "capabilities." The capability is usually characterized by the ability to perform a certain function or functions, and/or to provide a particular "interface" (which defines a set of functions together with the associated messages for requesting them). The capabilities may be annotated with options and constraints.

- The knowledge base typically groups resources into "domains" – named collections of resources that have some important properties in common, such as physical location, network access, security policies, or payment schemes. A given resource may belong to multiple such domains. The broker's knowledge base is usually *the* repository for all the resources in a certain set of domains (or a certain intersection of domains), but a broker may also have access to other brokers or other repositories that are responsible for other domains.

- The knowledge base associates the resource or the (resource, capability) pair with one or more access mechanisms and the associated protocol rules. It may also provide a mapping between the elements of those protocols and a common reference model, or mappings between those protocols and some equivalent protocols for the same functions.

- The broker supports one or more protocols for making and responding to the "find" requests. The protocol typically allows the client to specify capabilities in terms of names and the presence/absence of certain options, and to limit the collection of admissible resources by domain or other characteristics.

- The broker may serve only as a finder, and end its transaction with the client by providing the access information for the requested capability from the selected resource(s). Alternatively, the broker may serve as an intermediary between the client and the selected resource: The broker may accept explicit requests for the defined services of the resource directly from the client, communicate those requests to the resource using the access protocols specified in the knowledge base, and return the results to the client using the client-broker protocol. That is, the broker may provide the additional service of protocol conversion between the client and the resource, so that the client need only know the protocol for communicating with the broker.

The model described above is found in the ISO Trader service and protoeol specifications [62], but similar models are specified in commercial broker service architectures, such as the one proposed by the Telecommunications Information Networking Architecture Consortium (TINA-C) [106]. Both the finder function and the protocol conversion function may be important services in automating integration.

## 6.11 Code generation and reuse

Tools that generate code or retrieve appropriate code from libraries to solve particular technical problems in programming could be helpful in automating the solution of analogous technical problems in integration.

### 6.11.1 Middleware

If systems to be integrated are not already configured to communicate with one another, the task of getting them to communicate may be simplified by middleware.

Middleware supplies a reusable communication infrastructure that is useful for integrating programs. At a minimum, it includes some form of remote invocation or messaging, but may also include features designed to bridge between different operating systems, programming languages, communications protocols, coordination models, and other differences in the technical "platform." Different instances of middleware differ in the level of transparency that they provide at the source code level. They are distinguished from standard interchange formats or standard communications protocols in that some reusable component or service that implements the low-level details is included.

In CORBA [84], the goal is to provide enough transparency that interacting with a remote service is not much different than interacting with a local program object. This is accomplished by generating code for local objects that hide the communication with the remote service. In other middleware technologies the act of communicating is not hidden. A program must invoke a standard API to communicate, but the technical details of the communication itself are still handled by the middleware.

Other middleware technologies include the Distributed Computing Environment (DCE) [107], assorted services that are bundled with Java [127], and .Net [109].

### 6.11.2 Component architectures

Component toolkits range from simple libraries of reusable code to "program synthesis" tools, which compose programs to meet a formal specification by generating linkages among selections from a library of formally described components. An example of the latter is Amphion [76]. The same technology would be required in self-integration to compose a functioning system that meets some formally specified requirements. However, successful examples of program synthesis generate only a narrow range of programs, and even this requires a large investment of time in creating, verifying, and validating formal specifications.

# 7 Summary and conclusions

Integration is a software engineering discipline that has significant overlaps with systems engineering. The first step in approaching an integration problem is to recognize that it is a systems design problem and the result will be a new system. That means that it is most important to formulate the overall requirements for the new system and use those to drive all the design choices. In a similar way, machine-interpretable formulations of requirements will be critical to the automation of any integration task.

This paper describes the elements of the engineering process that is called "integration" and identifies those in which significant automation may be possible. In general, those elements are the "data integration" activities and the "technical integration" activities.

This paper identifies a set of common conflicts and concerns encountered in solving an integration problem. Again, those that may be addressed by automated approaches tend to be technical conflicts or simple semantic conflicts.

This paper identifies many technologies that might be brought to bear on the automation of the engineering tasks involved in integration, and other techniques that provide practical engineering approaches. Chief among these are:

- the available formal specifications for existing software systems

- emerging technologies for the capture and recognition of semantic descriptions of systems and data
- mechanisms for automated reasoning of several kinds
- mechanisms for integrating semantic information from multiple sources and recognizing commonalities

We undertook this study to determine whether the time is ripe for the application of advanced information technologies to solving the real integration problems. We believe that there are opportunities in this area, but further experiments involving the use of these technologies to solve some actual industrial integration problems are necessary to complete that determination, identify the hard problems, and evaluate the proposed solutions.

To root the integration problems in real-world concerns and issues, industry-relevant reference scenarios are needed to validate approaches and candidate solutions. In this context, integration scenarios are descriptions of the pertinent business processes and objectives, required information flows, and the application software systems and information resources available for integration into the target system.

An experiment would then involve:

- Developing tools that use proposed methods and technologies to perform some part of the integration process, using specifications for the requirements and specifications for the available resources; for example, the tool might create the links and wrappers needed to connect two or more available resources
- Formalizing the actual requirements for the integrated system per some industry scenario
- Obtaining and completing the data models and interface models for the available resources, and possibly converting those models to a form useable by the tools
- Capturing the relationships between the interface elements and the business processes being automated
- Running the integration tools and evaluating the results, including both the effectiveness of the tool in supporting the integration process and the amount of human effort invested in converting the problem-specific information to the forms needed by the integration tools.

We expect further research to include such experiments.

## Acknowledgements

The AMIS project team wishes to thank Sidney Bailin, Michael Gruninger and Vincent Barrilliot for the insight and expertise they contributed to the content of this report.

## References

[1] Batini, C.; Lenzerini, M.; "A Methodology for Data Schema Integration in the Entity Relationship Model," *IEEE Transactions on Software Engineering*, 10(6), 1984 November, pp. 650-663.

[2] Batini, C.; Lenzerini, M.; Navathe, S.B.; "A Comparative Analysis of Methodologies for Database Schema Integration," *ACM Computing Surveys*, 18(4), 1986 December, pp. 323-364.

[3] Bézivin, Jean; "Who's Afraid of Ontologies?," OOPSLA'98 Workshop #25, 1998. Available at http://www.metamodel.com/oopsla98-cdif-workshop/bezivin1/.

[4] Breitbart, Y.; Olson, P.L.; Thompson, G. R.; "Database Integration in a Distributed Heterogeneous Database System," in *Proceedings of the Second IEEE Conference on Data Engineering*, 1986 February.

[5] Calvanese, D.; De Giacomo, G.; Lenzerini, M.; "Ontology of Integration and Integration of Ontologies," in *Working Notes of the 2001 International Description Logics Workshop (DL-2001)*, Stanford, USA, 2001 August.

[6] Dalianis, H.; "The VINST approach: Validating and Integrating STEP AP Schemata Using a Semi Automatic Tool," in *Proceedings of the Conference on Integration in Manufacturing*, 1998 October.

[7] Dalianis, H.; Hovy, E.; "Integrating STEP Schemata using Automatic Methods," in *Proceedings of the ECAI-98 Workshop on Applications of Ontologies and Problem-Solving Methods*, 1998 August, pp. 54-66.

[8] Date, C.J.; *An Introduction to Database Systems*, Vol. I, 5th edition, Addison-Wesley, 1990.

[9] Elmasri, R.; Larson, J.; Navathe, S.B.; *Schema Integration Algorithms for Federated Databases and Logical Database Design*, Honeywell Corporate Systems Development Division, Report CSC-86-9:8212, 1986 January.

[10] Elmasri, R.; Navathe, S.B.; "Object Integration in Database Design," in *Proceedings of the IEEE COMPDEC Conference*, 1984 April.

[11] Fankhauser, P.; Kracker, M.; Neuhold, E.; "Semantic vs. Structural Resemblance of Classes," *SIGMOD Record*, 20(4), 1991 December, pp. 59-63.

[12] Federal Information Processing Standard Publication 183, *Integration Definition for Functional Modeling (IDEF0)*, National Institute of Standards and Technology, 1993 December, available from National Technical Information Service, U.S. Department of Commerce, Springfield, VA.

[13] Federal Information Processing Standard Publication 184, *Integration Definition for Information Modeling (IDEF1-X)*, National Institute of Standards and Technology, 1993 December, available from National Technical Information Service, U.S. Department of Commerce, Springfield, VA.

[14] Fillion, F.; Menzel, C.; Blinn, T.; Mayer, R.J.; "An Ontology-Based Environment for Enterprise Model Integration," in *Proceedings of the IJCAI Workshop on Basic Ontological Issues in Knowledge Sharing*, 1995 August.

[15] Flater, D.; "Impact of Model-Driven Standards," in *Proceedings of the 35th Hawaii International Conference on System Sciences*, 2002 January, CD-ROM file name DATA/STEAI01.PDF.

[16] García-Solaco, M.; Castellanos, M.; Saltor, F.; "Discovering Interdatabase Resemblance of Classes for Interoperable Databases," in *Proceedings of the IEEE RIDE-International Workshop on Interoperability in Multidatabase Systems*, 1993.

[17] Gruber, T.R.; "Toward principles for the design of ontologies used for knowledge sharing," in: Guarino, N. and Poli, R., eds., *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Kluwer, 1993.

[18] Hammer, J.; McLeod, D.; "An Approach to Resolving Semantic Heterogeneity in a Federation of Autonomous, Heterogeneous Database Systems," *International Journal of Intelligent and Cooperative Information Systems*, 2(1), 1993 March, pp. 51-83.

[19] ISO 10303-1:1994. *Industrial automation systems and integration — Product data representation and exchange — Part 1: Overview,* International Organization for Standardization, Geneva, Switzerland, 1994.

[20] ISO 10303-11:1994, *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: EXPRESS language reference manual,* International Organization for Standardization, Geneva, Switzerland, 1994.

[21] ISO DIS 10303-14:2001, *Industrial automation systems and integration — Product data representation and exchange — Part 14: Description methods: EXPRESS-X language,* International Organization for Standardization, Geneva, Switzerland, 2001.

[22] Kashyap, V.; Sheth, A.; *Schema Correspondences between Objects with Semantic Proximity*, Technical Report DCS-TR-301. Dept. of Computer Science, Rutgers University, October 1993.

[23] Kashyap, V.; Sheth, A.; "Schematic and Semantic Similarities between Database Objects: A Context-based Approach," *Very Large Data Bases (VLDB) Journal*, 5(4), 1996, pp. 276-304.

[24] Kashyap, V. and Sheth, A.; "Semantic Heterogeneity in Global Information Systems: The Role of Metadata, Context and Ontologies," in: Papazoglou, M.P. and Schlageter, G., eds., *Cooperative Information Systems: Current Trends and Directions*, Academic Press, 1998, pp. 139-178.

[25] Kim, W.; Choi, I.; Gala, S.; Scheevel, M.; "On Resolving Schematic Heterogeneity in Multidatabase Systems," *Distributed and Parallel Databases*, 1, 1993, pp. 251-279.

[26] Larson,J.A.; Navathe, S.B.; Elmasri, R.; "A Theory of Attribute Equivalence in Databases with Application to Schema Integration," *IEEE Transactions on Software Engineering*,15(4), 1989 April, pp. 449-463.

[27] Lenat, D. B. ; Guha, R. V.; Pittman, K.; Pratt, D.; Shepherd, M.:"Cyc: Toward Programs with Common Sense." *Communications of the ACM*, 33(8), 1990 August, pp. 30-49.

[28] Minsky, Marvin L. ; "Neural Nets and the Brain Model Problem," Ph.D. dissertation in Mathematics, Princeton University, 1954.

[29] Minsky, Marvin L. ; "Some Universal Elements For Finite Automata," in *Automata Studies*, Shannon, C.E., and McCarthy, J., eds., Annals of Mathematics Studies, Volume 34, Princeton University Press, 1956.

[30] Mitra, p.; Wiederhold, Gio; Jannink, J.; "Semi-automatic Integration of Knowledge Sources," in *Proceedings of Fusion '99*, July 1999.

[31] Modell, Martin E.: *A Professional's Guide to Systems Analysis*, 2nd. Ed., McGraw-Hill, 1996.

[32] Navathe, S. B.; Sashidhar, T.; Elmasri, R.; "Relationship Merging in Schema Integration," in *Proceedings of the 10th International Conference on Very Large Databases*, 1984.

[33] *New Shorter Oxford English Dictionary*, 1993 edition, Oxford University Press, 1993.

[34] Papakonstantinou, Y.; Garcia-Molina, H.; Widom, J.; "Object Exchange Across Heterogeneous Information Sources," in *Proceedings of the IEEE International Conference on Data Engineering*, Taipei, Taiwan, 1995 March, pp. 251-260.

[35] Peterson, J.L.; "Petri Nets," in *ACM Computing Surveys*, 9(3), 1977 September.

[36] Rational Software Corporation, "Issues using Model Integrator to merge 'unrelated' models," Technote 7382, 1999-04-23. Available at http://www.rational.com/technotes/rose_html/Rose_html/technote_7382.html.

[37] Rational Software Corporation, Rational Rose product, http://www.rational.com/products/rose/.

[38] Sheth, A. P. ; Gala, S. K.; Navathe, S. B.; "On automatic reasoning for schema integration," *International Journal of Intelligent and Cooperative Information Systems*,2(1), 1993, pp. 23-50.

[39] Sheth, A.; Kashyap, V.; "So Far (Schematically) yet So Near (Semantically)," in *Proceedings of the IFIP TC2/WG2.6 Conference on Semantics of Interoperable Database Systems*, DS-5, 1992 November. Also in *IFIP Transactions A-25*, North Holland, 1993.

[40] Yu, C.; Sun, W.; Dao, S.; Keirsey, D.; "Determining relationships among attributes for Interoperability of Multidatabase Systems," in *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, 1991 April.

[41] Aggarwal, C.C.; Al-Garawi, F.; Yu, P.S.; "On the Design of a Learning Crawler for Topical Resource Discovery," *ACM Transactions on Information Systems*, 19(3), 2001, pp. 286-309.

[42] Borgida, A.; Devanbu, P.T.; "Adding More 'DL' to IDL: Towards More Knowledgeable Component Inter-Operability," in *Proceedings of the 1999 International Conference on Software Engineering*, 1999, pp. 378-387.

[43] Davis. R.; Smith. R.G.; "Negotiation as a Metaphor for Distributed Problem Solving," *Artificial Intelligence*, 20(1), 1983, pp. 63-103.

[44] Duffy, David; *Principles of Automated Theorem Proving*, Wiley, 1991.

[45] Foundation for Intelligent Physical Agents (FIPA) Contract Net Interaction Protocol Specification, available at http://www.fipa.org/specs/fipa00029/.

[46] Jones, Cliff B. ; *Systematic Software Development Using VDM*, 2nd edition, Prentice Hall, 1990.

[47] Schwartz, Michael F. ; "The Networked Resource Discovery Project," in *Proceedings of the IFIP XI World Congress*, 1989 August, pp. 827-832.

[48] Schwartz, Michael F. ; *The Networked Resource Discovery Project: Goals, Design, and Research Efforts*, technical report CU-CS-387-88, Department of Computer Science, University of Colorado, May 1988.

[49] Spivey, J. M.; *The Z Notation: A Reference Manual*, 2nd edition, Prentice Hall, 1992.

[50] Weibel, S.; Kunze, J.; Lagoze, C.; Wolf, M.; "Dublin Core Metadata for Resource Discovery." Internet Engineering Task Force RFC 2413, 1998 September.

[51] Garlan, David ; "Foundations for Compositional Connectors." position paper for the International Workshop on the Role of Software Architecture in Testing and Analysis, 1998. Available at http://www.ics.uci.edu/~djr/rosatea/papers/garlan.pdf.

[52] ARIANE 5 Flight 501 Failure, Report by the Inquiry Board, 1996. Available at http://www.esa.int/export/esaCP/Pr_33_1996_p_EN.html.

[53] Flater, David: "Testing for Imperfect Integration of Legacy Software Components," in *Proceedings of the 1st Asia-Pacific Conference on Quality Software*, IEEE Computer Society, 2000 October, pp. 156-165.

[54] Garlan, D.; Allen, R.; Ockerbloom, J.; "Architecture Mismatch: Why Reuse is so Hard," *IEEE Software*, 12(6), 1995 November, pp. 17-26.

[55] Garlan, D.; and Shaw; "An Introduction to Software Architecture," in *Advances in Software Engineering and Knowledge Engineering, vol. I*, World Scientific Publishing Company, 1993.

[56] Brousell, D.; "The Integrated Enterprise Moves Closer to Reality," in *Managing Automation*, 16(10), 2001 October. Thomas Publishing Co, New York, NY, pp. 26-30.

[57] Brunnermeier, S.; Martin, S.; "Interoperability Cost Analysis of the U.S. Automotive Supply Chain," Research Triangle Institute, March 1999.

[58] National Coalition for Advanced Manufacturing (2001), *Exploiting E-Manufacturing: Interoperability of Software Systems Used by U.S. Manufacturers*, Washington, D.C., NACFAM.

[59] ISO/IEC 10746-1:1998 *Information technology – Open Distributed Processing: Reference Model – Part 1: Overview*, International Organization for Standardization, Geneva, Switzerland, 1998.

[60] ISO/IEC 9075-2:1999, *Information technology – Database languages SQL – Part 2: Foundation*, International Organization for Standardization, Geneva, Switzerland, 1999.

[61] ISO/IEC 14750:1999, *Information technology – Open Distributed Processing – Interface Definition Language*, International Organization for Standardization, Geneva, Switzerland, 1999. Also available from the Object Management Group as: "IDL Syntax and Semantics," Chapter 3 in *Common Object Request Broker Architecture (CORBA) version 3.0*, http://www.omg.org/cgi-bin/doc?formal/02-06-39.

[62] ISO/IEC 13235-1:1998, *Information technology – Open Distributed Processing – Trading function: Specification*, International Organization for Standardization, Geneva, Switzerland, 1998,

[63] Brousell, D. "The Integrated Enterprise Moves Closer to Reality," *Managing Automation*, 16(10), Thomas Publishing Co, New York, 2001 October, pp 26-30.

[64] *Industry Leaders and Manufacturing Excellence: The CASA/SME Industry LEAD Award Winners (1981-2001)*, The Computer and Automated Systems Association of the Society of Manufacturing Engineers, 2001.

[65] Quine, W. V.; *From a Logical Point of View: Nine Logico-Philosophical Essays*, Harvard University Press, 1980.

[66] IEEE-Std-1471-2000 *Recommended Practice for Architectural Descriptions for Software-Intensive Systems*, Institute for Electrical and Electronics Engineering, New York, NY, 2000.

[67] Minsky, M. L.; *The Society of Mind*, Simon & Schuster, New York, 1986.

[68] Jackson, Michael; *Software Requirements and Specifications*, Addison Wesley, 1995.

[69] Zave, P.; Jackson, M.; "Four Dark Corners of Requirements Engineering," *ACM Transactions on Software Engineering and Methodology*, 6(1), 1997 January, pp. 1-30.

[70] Pollock, Jeffrey T., "Interoperability vs. Integration," in *eAI Journal*, 2001 October. Available at: http://www.eaijournal.com/Article.asp?ArticleID=441 or http://www.eaijournal.com/PDF/IntvsIntPollock.pdf

[71] Pinkston, Jeff, "How EAI Differs from B2B," in *eAI Journal*, 2001 August. Available at: http://www.eaijournal.com/Article.asp?ArticleID=403 or http://www.eaijournal.com/PDF/Ins&OutsPinkston.pdf

[72] Cadarette, Paul and Durward, Karen, "Achieving a Complete Enterprise Integration Strategy," EbizQ, 2001 December. Available at: http://eai.ebizq.net/str/cadarette_1.html

[73] Berners-Lee, T.; *Weaving the Web*, Harper, San Francisco, 1999 September.

[74] McIlraith, Sheila A.; "Semantic Web Services," *IEEE Intelligent Systems*, 16(2), 2001 March/April.

[75] The DAML Services Coalition, "DAML-S: Semantic Markup For Web Services," in *Proceedings of the International Semantic Web Working Symposium (SWWS)*, July 30-August 1, 2001. Available at: http://www.daml.org/services/SWWS.pdf

[76] Lowry, M.; Philpot, A.; Pressburger, T.; Underwood, I.: "Amphion: Automatic Programming for Scientific Subroutine Libraries", in *Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems*, Charlotte, North Carolina, Oct. 16-19, 1994, Springer-Verlag Lecture Notes in Computer Science, Vol. 869, pp. 326-335. Available via the Amphion Project Page: http://ase.arc.nasa.gov/docs/amphion.html.

[77] Corcho, C.; Gómez-Pérez, A.; "A Roadmap to Ontology Specification Languages," in *Proceedings of the 12th International Conference on Knowledge Engineering and Knowledge Management*, October 2000, pp. 80-96.

[78] Clark, James, ed., *XSL Transformations (XSLT) Version 1.0*,W3C Recommendation 16 November 1999, WorldWideWeb Consortium, 1999, Available at http://www.w3.org/TR/xslt.

[79] Weerawarana, S.; Chinnici, R.; Gudgin, M.; Moreau, J.-J.; eds.;*Web Services Description Language (WSDL) Version 1.2*, W3C Working Draft 9 July 2002, WorldWideWeb Consortium, 2002. Available at http://www.w3.org/TR/wsdl12.

[80] Object Management Group, *Unified Modeling Language (UML) version 1.4*, Object Management Group, Needham, MA, 2001. Available at http://www.omg.org/technology/documents/formal/uml.htm.

[81] Object Management Group, *XML Metadata Interchange (XMI), version 1.2*, Object Management Group, Needham, MA, 2001. Available at http://www.omg.org/technology/documents/formal/xmi.htm.

[82] Object Management Group, *Negotiation Facility, version 1.0*, Object Management Group, Needham, MA, 2001. Available at http://www.omg.org/technology/documents/formal/negotiation.htm.

[83] Object Management Group, *Meta-Object Facility (MOF), version 1.4*, Object Management Group, Needham, MA, 2001. Available at http://www.omg.org/technology/documents/formal/mof.htm

[84] Object Management Group, *Common Object Request Broker Architecture (CORBA) version 3.0*, Object Management Group, Needham, MA, 2001. Available at http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[85] American National Standard X3.138-1989, *Information Resource Dictionary Systems*, American National Standards Institute, New York, 1989.

[86] Gold-Bernstein, B., Marca, D., *Designing enterprise client/server systems*, Prentice Hall, 1998.

[87] Wenzel, B, "Introduction to ISO FDIS 10303-214," presentation to the Object Management Group, 2000 April.

[88] Albus, J.S., *A Reference Model Architecture for Intelligent Systems Design*, NISTIR 5502, National Institute of Standards and Technology, Gaithersburg, MD, 1994 September. Available at: http://www.isd.mel.nist.gov/documents/albus/Ref_Model_Arch345.pdf.

[89] Zeus Agent Building Toolkit: http://193.113.209.147/projects/agents/zeus/index.htm.

[90] Ross, R., *The Business Rule Book: Classifying, Defining and Modeling Rules*, 2nd Ed., Business Rule Solutions, LLC, 1997.

[91] Chen, P. P-S., "The Entity-Relationship model: Toward a unified view of data," ACM Transactions on Database Systems, 1(1), 1976, pp. 9-36.

[92] Rumbaugh, J, M.; Blaha, W.; Premerlani, F.E.; Lorensen, W.; *Object Oriented Modelling and Design,* Prentice Hall, 1991.

[93] Booch, G., *Object-Oriented Analysis and Design with Applications* (2nd ed.), Benjamin/Cummings, Redwood City, CA, 1994.

[94] Shlaer, S. ; Mellor, S, *Object-oriented systems analysis: modeling the world in data*, Yourdon Press, Englewood Cliffs, NJ, 1988..

[95] Jacobsen, I.; Christerson, M.; Jonsson, P.; Overgaard, G.; *Object-Oriented Software Engineering – A Use-case Driven Approach*, Addison-Wesley, 1992.

[96] ebXML, "ebXML Technical Architecture Specification v1.04," ebXML consortium, 2001 February. Available at: http://www.ebxml.org/specs/ebTA.pdf

[97] Harel, D.; "Statecharts: A Visual Formalism for Complex Systems," in *Science of Computer Programming*, 8(3), 1987 June, pp. 231-274.

[98] Moore, E. F.; "Gedanken experiments on sequential machines," in *Automata Studies*, 1956, pp. 129-153, Princeton University Press, NJ, USA.

[99] Halpin, T.A., *Information Modeling and Relational Databases*, Morgan Kaufmann, 2001.

[100] Nijssen, G.M.; Halpin, T.A.; *Conceptual Schema and Relational Database Design – A fact-oriented approach*, PrenticeHall, 1989.

[101] Meersman, Robert; "Towards Models for Practical Reasoning about Conceptual Database Design," in: Meersman, R., and Sernadas, A., eds., *Proceedings of the IFIP WG 2.6 Working Conference on Data Semantics, Data and Knowledge,* Albufeira, Portugal, November, 1986, North-Holland 1988, pp. 245-263

[102] International Organization for Standardization, *SUMM, Semantic Unification Meta Model*, ISO/IEC JTC1 SC2 WG3. N1360, 1991-Oct-15.

[103] Fulton, J. A., "Enterprise integration using the Semantic Unification Meta-Model," in: C. J. Petrie, Jr., ed., *Proceedings, 1st International Conference on Enterprise Integration Modeling*, pp. 278-89, 1992.

[104] "About the DAML [Defense Advanced Research Projects Agency (DARPA) Agent Markup Language] language." Available at: http://www.daml.org/about.html

[105] Lesser, Victor R.; "Multiagent Systems: An Emerging Subdiscipline of AI," *ACM Computing Surveys*, 27 (3). 1995 September,, pp. 340-342. Available via: http://portal.acm.org/citation.cfm?doid=212094.212121

[106] Kristiansen, L., Ed., Telecommunications Information Networking Architecture Consortium (TINA-C), *Service Architecture 5.0*, 1997 June. Available at: http://www.tinac.com/specifications/documents/sa50-main.pdf

[107] The Open Group, *DCE – Distributed Computing Environment: Overview*, The Open Group, Inc., Woburn, MA, 1996. Available at: http://www.opengroup.org/dce/info/papers/tog-dce-pd-1296.

[108] Microsoft Corp., *Distributed Component Object Model (DCOM)*, Microsoft Corporation, Seattle, WA, 1998. Available via: http://www.microsoft.com/com/tech/dcom.asp

[109] Microsoft Corp., *Microsoft .NET for IT Professionals*, Microsoft Corporation, Seattle, WA, 2002.. Available at: http://www.microsoft.com/technet/itsolutions/net/evaluate/itpronet.asp

[110] Gosling, J.; Joy, W.; Steele, G.; Bracha, G.; *Java Language Specification 2.0*, Addison-Wesley, 2000. Available at: http://java.sun.com/docs/books/jls

[111] Jacobs, Barry E.; *Applied Database Logic*, Prentiss-Hall, 1985.

[112] Smith, B.; Welty, C.; "Ontology: Towards a new synthesis," in: Welty, C. and Smith, B., eds., *Formal Ontology in Information Systems*, ACM Press, Ongunquit, Maine, 2001, pp. iii-x.

[113] Sacerdoti, E. D.; "The nonlinear Nature of Plan," in: Allen, J.; Hendler, J.; Tate, A.; editors, *Readings in Planning,* Morgan Kaufman, 1990, pp 162-170

[114] Tate, A; "Generating Project Networks," in: Allen, .J; Hendler, J; Tate, A; editors, *Readings in Planning,* Morgan Kaufman, 1990, pp 162-170

[115] Erol, K.; Hendler, J.; Nau, D. S. "HTN Planning: Complexity and Expressivity," in *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1994.

[116] ISO/IEC 7498-1:1994, *Information technology – Open Systems Interconnection — Basic Reference Model: The Basic Model*, International Organization for Standardization (ISO), Geneva, Switzerland. 1994.

[117] de Kleer, Johan, "An Assumption-Based TMS," Artificial Intelligence, 28(2): pp. 127-162, 1986.

[118] Object Management Group, "Model-Driven Architecture (MDA)," Object Management Group, Needham, MA, 2001. Available via http://www.omg.org/mda/specs.htm.

[119] ISO/IEC 8824-1:1998, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*, International Organization for Standardization (ISO), Geneva, Switzerland, 1998.

[120] Bray, T.; Paoli, J.; Sperberg-McQueen, C.M.; Maler, E.; eds.; *Extensible Markup Language (XML) Version 1.0*, W3C Recommendation 6 October 2000, WorldWideWeb Consortium, 2000. Available at http://www.w3.org/TR/REC-xml.

[121] Berners-Lee,T.; Fielding, R.; Frystyk, H.; *Hypertext Transfer Protocol -- HTTP/1.0*, Internet Engineering Task Force RFC 1945, 1996 May. Available at http://www.ietf.org/rfc/rfc1945.txt

[122] Klensin, J., Ed., *Simple Mail Transfer Protocol*, Internet Engineering Task Force RFC 2821, 2001 April. Available at http://www.ietf.org/rfc/rfc2821.txt

[123] Box, D. et al.; *Simple Object Access Protocol (SOAP) 1.1*, W3C Note 08 May 2000, WorldWideWeb Consortium, 2000. Available at http://www.w3.org/TR/SOAP.

[124] ISO/IEC 9075-3:1999, *Information technology – Database languages – SQL – Part 3: Call-Level Interface (SQL/CLI)*, International Organization for Standardization (ISO), Geneva, Switzerland, 1999.

[125] Open Applications Group, "Open Applications Group Integration Specification (OAGIS) v8.0." Available via: http://www.openapplications.org/downloads/oagis/loadform.htm.

[126] Wiltamuth, S.; Hejlsberg, A.; *C# Language Specification*, Microsoft Corporation, Seattle, WA, 2002. Available at: http://msdn.microsoft.com/library/en-us/csspec/html/CSharpSpecStart.asp

[127] DeMichiel, Linda, Ed., *Enterprise Java Beans 2.0*, Java Community Process, March, 2002. Available via: http://java.sun.com/products/ejb/docs.html

[128] Berners-Lee, T.; Fielding, R.; Masinter, L.; *Uniform Resource Identifiers (URI): Generic Syntax*, Internet Engineering Task Force RFC 2396, 1998 August. Available at: http://www.ietf.org/rfc/rfc2396.txt

[129] Sowa, J.F., "Conceptual graphs for a database interface," *IBM Journal of Research and Development* 20(4), pp. 336-357.

[130] Klyne, G.; Carroll, J.; eds.; *Resource Description Framework (RDF): Concepts and Abstract Data Model*, W3C Working Draft 29 August 2002, WorldWideWeb Consortium, 2002. Available at: http://www.w3.org/TR/rdf-concepts/

[131] Lassila, O.; Swick, R.R.; eds.; *Resource Description Framework (RDF) Model and Syntax Specification*, W3C Recommendation 22 February 1999, WorldWideWeb Consortium, 1999. Available at http://www.w3.org/TR/REC-rdf-syntax/

[132] Brickley, D.; Guha, R.V.; eds.; *RDF Vocabulary Description Language 1.0: RDF Schema*, W3C Working Draft 30 April 2002, WorldWideWeb Consortium, 2002. Available at http://www.w3.org/TR/rdf-schema/

[133] Hayes, P., Ed.; *RDF Model Theory*, W3C Working Draft 29 April 2002, WorldWideWeb Consortium, 2002. Available at http://www.w3.org/TR/rdf-mt/

[134] Horrocks, I., et al., *The Ontology Inference Layer OIL*. Available via: http://www.ontoknowledge.org/oil/

[135] van Harmelen, F.; Patel-Schneider, P.F.; Horrocks, I.; eds.; *DAML+OIL (March 2001) Reference Description* W3C Note 18 December 2001 Available at:http://www.w3.org/TR/daml+oil-reference

[136] Stein, L.A., Connolly, D., and McGuinness, D., eds.; *Annotated DAML Ontology Markup,the DAML consortium*, 2000 March. Available at: http://www.daml.org/2000/10/daml-walkthru

[137] Sowa, J.F., *Knowledge Representation: Logical, Philosophical and Computational Foundations*, Brookes/Cole, 2000, pp. 476-489.

[138] Karp, P.D., *The design space of frame knowledge representation systems*, Technical Report 520, SRI International AI Center, 1992.

[139] von Halle, B., "Back to Business Rule Basics," *Database Programming & Design*, October 1994, pp. 15-18.

[140] Bailin, S.; Truszkowski, W.; "Ontology negotiation between intelligent information agents," in *Knowledge Engineering Review*, 17(1), 2002 March.

[141] Bailin, S.; "Object oriented analysis," in Marciniak, J. ed.; *Encyclopedia of Software Engineering*, 2nd Edition, John Wiley and Sons, 2002.

[142] Bailin, S.; Simos, M.; Levine, L.; Creps, R;. *Learning and Inquiry Based Reuse Adoption (LIBRA)*, IEEE Press/Wiley, 2000.

[143] Bailin, S.; "Software development as knowledge creation," *International Journal of Applied Software Technology*, 3(1), 1997 March.

[144] Bailin, S.; Truszkowski, W.; "Ontology negotiation: a dynamic approach to substantive communication between agents," in *Proceedings of the 5th World Multi-Conference on Systemics, Cybernetics and Informatics*, Orlando, Florida, 2001 July.

[145] Fellbaum, C., Ed.; *WordNet: An Electronic Lexical Database*, MIT Press, 1998 May.

[146] Gruninger, M.; Uschold, M.; "Ontologies and Semantic Integration," to appear in *Software Agents for the Warfighter*, Information Technology Assessment Consortium, 2002.